EB 145

HELIX Environment
Meta-Programmer Manual.

V.H. Coonen
July 25, 1988

Masters thesis on a project
done at the Philips Research
Laboratories in Eindhoven
under supervision of:

Ir. R. Schutten
Prof. Dr. Ir. C.J. Koomen

The department of electrical engineering of the Eindhoven University of Technology does not accept any responsibility for the contents of student reports and masters theses.

Abstract

The Behavioral Simulator Helix supports simulation of hierarchically-specified designs. The Helix System generates a simulator from the structure (Schematic) and the behavior (Models) of a design. The schematics are created with the Structural Description Language (SDL), the models are described in the Hierarchical Hardware Description Language (HHDL). This HHDL is a high-level language based on the Pascal programming language, but specially adapted for the simulation of integrated circuits.

A programming environment is created for the Helix System especially for HHDL structure support. The environment is generated from an environment specification by the environment generator PHIL. This specification consists of three layers : an *object*-layer, a *context-sensitive*-layer, and an *interaction*-layer. The object layer defines the objects that can be present in the created object-world. In all Phil environments there is a notion of structure. Objects are represented by means of a tree, the so called object-world. In the context-sensitive layer the invariant relations between objects are expressed by means of the definition of evaluators. In the interaction layer some representations and command are defined to interact with the objects present in the world.

The Phil Environment Generator seems a very flexible tool for generating environments. BNF specifications are easily transformed into the generator specification language Philgram. Relations between objects are easily defined by means of Philgram and the programming language Lisp. A predefined interaction facility, the Phil House Style (PHS) is a good basis for interaction with the defined object world. Only few application specific commands and representations have to be defined.

The Phil Environment Generator is built on Lisp and works on a SUN workstation, a SYMBOLIC workstation or a VAX/VMS machine.

Preface

This report deals with the final year project done as a conclusion of my study at the department of Electrical engineering at the Eindhoven University of Technology. The graduate work has been carried out at the Philips Research Laboratories Eindhoven from october 1987 until july 1988.

I would like to thank Prof. Dr. Ir. C.J. Koomen for offering me the oppportunity to do this project at the Philips Research Laboratories and for his support during the project.

Furthermore, I would like to thank Drs. L. Helmink, Ir. J.A. Droppert and Ing. M.J. van Tien for the helping hand they have offered me with some Lisp and Phil generator problems.

My special appreciation goes to my coach, Ir. R. Schutten for many useful suggestions and discussions and for the support he has given me during the project.

Vincent Coonen, July 1988

Contents

1 Introduction **1**

2 An Introduction to Helix **3**

3 The Environment-Generator PHIL **6**

3.1 An Introduction to Phil 6

3.2 The Object Layer 8

3.2.1 Constant definition 9

3.2.2 Primitive definition 9

3.2.3 Structure definition 10

3.2.4 Sequence definition 10

3.2.5 Sort definition 11

3.3 Class properties. 11

3.3.1 Predefined Sorts and Classes 12

3.4 The Context Layer 13

3.4.1 Referring to other objects 13

3.4.2 Creating new objects 15

3.4.3 Side Effects in Evaluators 16

3.4.4 Special Classes 18

3.5 The Interaction Layer 20

3.5.1 Interaction model 20

3.5.2 Format Specification 21
3.5.3 Defining Commands 27
3.5.4 The Phil House Style 27
3.5.5 Accessing panes and devices 29

4 The Helix Object Layer **31**

4.1 A Top Level definition. 31
4.2 Standard Routines. 31
4.3 Predefined Packages. 32
4.4 Ordinal Types. 32
4.5 Comment in the helix environment. 33
4.6 Unlabelled Statements. 33
4.7 Redefining some Sorts. 33
4.8 Procedure and Function declaration. 34
4.9 BNF Transformations. 34
4.10 Parser Optimizations 34

5 The Helix Context Layer **36**

5.1 Performing Checks using Evaluators 36
5.2 Type Specification Checking 38
5.3 Type Checking 38
5.3.1 Determining the type of a HHDL variable. 40
5.4 Storing Declarations using Symlayers 41
5.4.1 Determining Scopes 42
5.4.2 The Method :ptype 47
5.5 Double Definitions Checking 47
5.6 Interpackage Communication 48
5.7 Using Predefined Packages 49

5.8 Special Symlayers 50
5.8.1 Forward Symlayers 50
5.8.2 Assignment to Function variables 50
5.9 Phil Redefinitions 51
5.9.1 Redefining the function (RE)SET-COLOR 51
5.9.2 Redefining the macro NEW 51
5.9.3 Redefining the macro DEF-CONSTANT 51
5.10 Phil Bugs 52
5.10.1 Auto Frame 52
5.10.2 Eval 52
5.10.3 Retrieve 52

6 The Helix Interaction Layer 53

6.1 Interaction Alternatives 53
6.2 An Experienced User 54
6.3 An Inexperienced User 54
6.3.1 An Environment User Manual 54
6.3.2 Learning Helix 54
6.4 Creating a Helix pane 55
6.5 Helix Commands 56
6.5.1 The Manual Command 56
6.5.2 The Get-Type Command 57
6.6 Helix Modes 57
6.6.1 :short 57
6.6.2 :medium 57
6.6.3 :default 57

7 Conclusions and Suggestions 58

A Philgram 60

B Helix Syntax Specification 65

C The Method :ptype. 79

D Typed Objects. 81

E The :ctype Methods. 83

F The :vtype Methods. 84

G The :type Methods. 85

H Type specification checks 88

I Helix Context Checks 90

J checks-lisp 95

K default-plane 102

Chapter 1

Introduction

The Helix System is a fully portable software system and consists of a number of integrated programs used in the design and simulation of electronic circuits. The structure and behavior for a design are entered separately into the Helix System.
Structural information for a design is entered with the so called SDS tools. The behavior for each symbol used in a circuit is described with Hierarchical Hardware Description Language (HHDL). HHDL is a structured high-level language based on the Pascal programming language, but specifically adapted for the simulation of integrated circuits. HHDL models are presently created in a conventional text-editor. The source-program generated is processed by the HHDL-compiler (SIMCHK) which checks the HHDL syntax, declarations, consistency of use, and data type compatability.

An environment could be created to support the design process of electronic circuits. A software engineering environment (SEE) is the collection of facilities available to a designer, or a group of designers, to create and manipulate descriptions.

The basic idea of an environment is to incorporate as much knowledge as possible in this tool, which will then be available to the user. Two basic approaches to develop a SEE are mentioned. A dedicated approach and an approach based on generation.

In a dedicated approach in developing a SEE, the information on what constitutes the design method, and how design descriptions can, or have to be manipulated, are scattered trough the implementation of the SEE. This leads to environments, although often fairly efficient, that are rather inflexible, since a minor modification in the design method can lead to major modifications in the implementation of the environment. Hence this approach seems only viable when the design method is well established and few modifications are expected.

The generation approach is based on the principle that an environment is generated based on a *specification* of a design method. Here the basic manipulation facilities will be provided by the generator kernel, which then can be specialised for design method dependent manipulation facilities. By means of a generator a virtual unlimited amount of design methods can be *implemented.* Further, modifications and extentions are a lot easier to implement, since they are only to be dealt with on the level of the environment specification rather than the implementation.

The Environment Generator PHIL is fed by a specification in a formal language, PHILGRAM, and it then produces a PHIL environment. To facilitate the tasks of a specification, different layers can be distinguished in Philgram, each with its own meaning:

- ☐ structure layer
- ☐ context sensitive layer
- ☐ interaction layer

The Philgram structure layer consists of a set of class definitions which serve as blueprints, or templates, for their instances, the objects.
On top of the structure layer lies the context sensitive layer. The evaluators in the class definitions express the dependencies between objects. In this layer, the system invariance is formalized, which incorporates the type correctness as well as context sensitive constraints.
In the interaction layer the interaction with the Phil-world of objects is organized. Commands and representations can be specified in this layer.

The Helix environment should be able to speed up the HHDL model design process, because model descriptions created in this SEE are always syntactically correct. The environment could also be used to teach the environment user, how create HHDL model descriptions. An on-line environment user manual would also be a convenient function of the environment. The contents of the interaction layer depends on the the kind of user that will eventually use the environment. E.g. inexperienced and experienced Helix users need different interaction facilities, and therefore a different interaction layer. The support given to user must be defined in the interaction layer of the environment specification.

The generator PHIL is fully described in chapter 3. Specification of the three layers for the HELIX application can be found in the chapters 4, 5 and 6.

Chapter 2

An Introduction to Helix

The Behavioral Simulator Helix supports simulation of hierarchically-specified designs. Structure (Schematic) and Behaviour (Models) for a design are entered separately into the HELIX system. See figure 2.1.

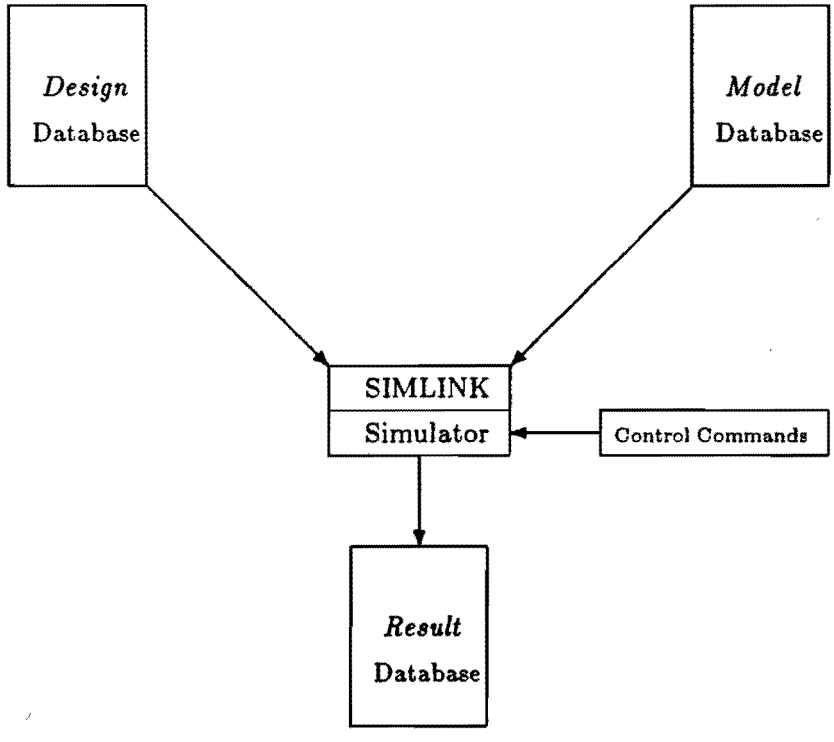

Figure 2.1: The HELIX System.

All models are created by the user or are extracted from standard model libraries. All models for the Helix system are encoded in HHDL (Hierarchical Hardware Description Language). The major part of generating a helix sim-

ulator is to describe models in this description language HHDL, which is an extended PASCAL version. Normally the model description created by the user is compiled and a simulator is returned. If the description contains some errors, the source file cannot be compiled and some error information is returned. The user will go on editing the model description and will invoke the compiler again. After some attempts the model description is syntactically correct and a simulator is generated by the Helix system. Especially inexperienced designers are forced to re-edit the source over and over before it is syntactically correct.

An example of a HHDL model description is presented to the reader. Consider the two-bit adder of figure 2.2.

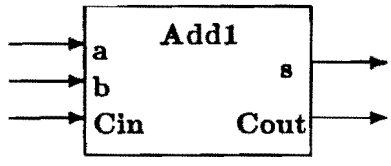

Figure 2.2: A two-bit adder.

The HHDL model description could be something like:

```
Module Adder;

Nettype Logic = Boolean;

Comptype Add1;
Inward a,b,Cin : Logic;
Outward s,Cout : Logic;

Subprocess

    Add_carry_in    : Upon Cin=true Check a,b,Cin Do
                        begin
                          assign (a = b) to s delay 1;
                          assign (a OR b) to Cout delay 1;
                        end;

   Add_no_carry_in : Upon Cin=false Check a,b,Cin Do
                        begin
                          assign (a = b) to s delay 1;
                          assign (a OR b) to Cout delay 1;
                        end;

Begin
end;
```

The example presented gives an impression of a model description in HHDL. Models must be described in the *Comptype*-part of a HHDL *Module*. Three pins *a,b,Cin* are defined to be input pins. The pins *s* and *Cout* are output pins. All the pins defined are of a certain *Nettype*, the nettype *Logic* which was defined in the *Nettype* declaration part as the type *Boolean*. The *Subprocesses* describe the actual behavior of the Adder circuit. Two subprocesses are defined. The *Upon-Do* construction is used to trigger actions on changes of pins. So if one of the *inward*-pins changes and the carry-in is true, then a new value is assigned to the *outward*-pins *s* and *Cout*. The second subprocess describes the situation if the carry-in is false. Values are assigned to the output-pins with a delay of 1 clock-period.

The complete syntax specification of HHDL is given by [5]. This specification must be transformed into a PHILGRAM specification, before it can be fed to the environment-generator PHIL. Philgram will be fully explained in the following chapter. The Helix Layers are described in the chapters 4, 5 and 6.

Chapter 3

The Environment-Generator PHIL

A software engineering environment is the collection of facilities available to a designer, to create and manipulate descriptions. Phil is an environment generator which can generate an environment from an environment specification. The input of the PHIL generator is a specification in PHILGRAM. Philgram specifications consist of a set of class definitions. Class definitions serve as blue-prints, or templates, for their instances, the objects. One of the aspects of a class definition is the definition of methods. Each object is an instance of a class and has the methods of this class. The methods, or private functions, of an object determine the reaction of an object on receiving messages. If a method corresponding to the received message exists, then this method is executed and the result is returned to the sender of the message. The only possible interaction with objects is via sending messages. An object oriented approach results in

3.1 An Introduction to Phil

PHIL is an environment generator. This means that environments must be specified, and from this specification an environment is generated. This chapter describes how environments can be specified. Prerequisite for this report are the documents "An Abstract Functional Description of the PHIL Environment Generator", "General Concepts in PHIL Environments" and "Phil Definers manual". Further a working knowledge of Common Lisp is expected. Common to all Phil environments is a notion of structure. Instead of manipulating (editing) text, one manipulates (edits) structures. How one actually manipulates structures, is dependent on the interface defined. The structure that is edited in the environment, is a general tree structure. This tree will be referred to as the world. A subtree from this world will be referred to as an object. All manipulations performed on objects will ultimately result in the following operations :

- replacement of one object by another
- the addition of an object, or the removal of an object

To be able to manipulate structures there is a notion of a current (selected) object.

An environment specification contains three major layers:

- A specification of all classes which will be present in the intended environment. Instances of these classes will be referred to as objects. In this layer the logical structure of an environment will be specified, and will be referred to as the object layer of a specification.
- A specification of the relations between objects in the environment. In this layer is defined how objects are dependent on other objects. This will be referred to as the context layer of the specification.
- A specification of the interaction a user can have with the objects in the environment. This will be referred to as the interaction layer of the specification.

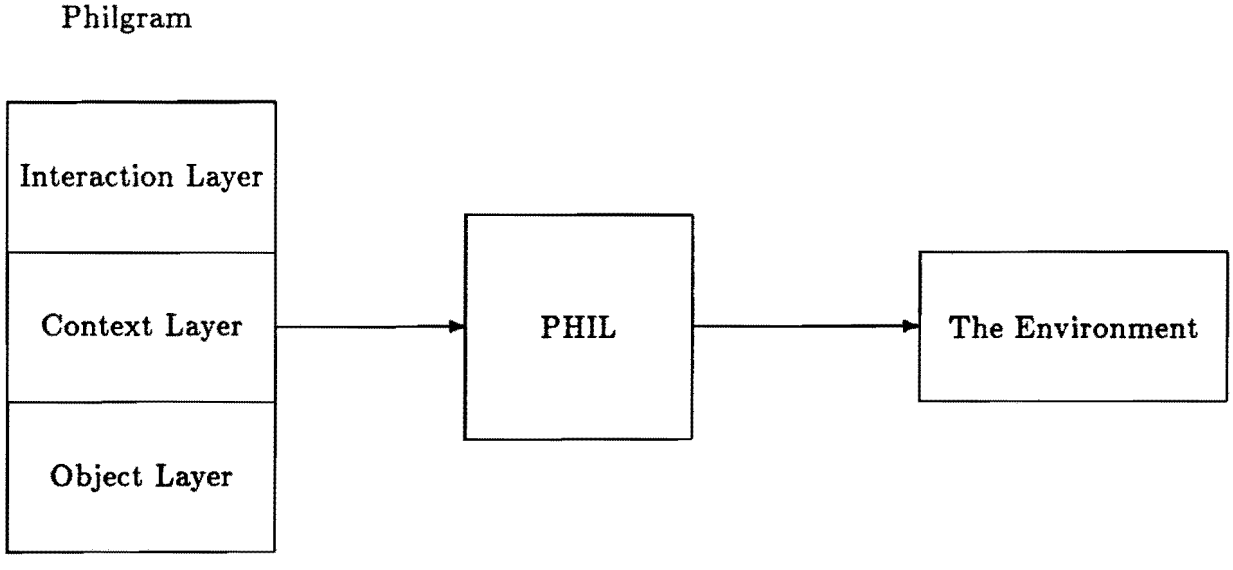

Figure 3.1: The Generator Phil.

These three layers will conceptually be present in any environment specification. However PHIL does not require that your application is modularized in accordance with this layering. A definer is free to use any kind of modularization he wishes.

3.2 The Object Layer

The logical structure of an environment is defined in the object layer of an environment specification. This structure is defined in terms of classes of objects. PHILGRAM supports the definition of four kinds of classes: constant, primitive, structure and sequence classes. Classes of kind constant or primitive will sometimes be referred to as a leaf classes. These four kinds of classes together with the notion of sorts, provide an expressive power which is equal to BNF. Hence general context free structures can be defined. The generic form to define classes is:

def-xxx ::= (**def-***xxx* *class-name-spec xxx-spec*
[**:doc** *documentation*]
[**:format** *xxx-format-spec*]
[**:methods** ({*method-spec*}*)])

documentation ::= STRING
class-name-spec ::= *class-name* | (*class-name* {*prop-name prop-value*}*)
class-name ::= SYMBOL
prop-name ::= KEYWORD
prop-value ::= FORM

where *xxx* will be replaced by the kind of class being defined. A *name-spec* specifies the name of the class being defined. This can either be SYMBOL, in which case it is the name of the class, or a form (*class-name* {KEYWORD FORM}*), where *class-name* again is the name of the class and {*prop-name prop-value*}* allows the definition of a set of *properties* for this class. If only a *class-name* is specified a set of default properties are assigned to the class. Class properties, and how to use them are discussed in the the "Phil Definers Manual". The documentation supplied after the :doc keyword will be used for brief on-line documentation. If the :doc option is not supplied PHIL will generate a short documentation string itself.

The class specifications for the four classes constant, primitive, structure and sequence will be discussed in the next four sections of this chapter. The format specifications for the classes are discussed in section 3.5.1 while the method specification is discussed in section 3.4.1.

Properties, format specifications and method specification can also be defined separatly from a class definition. How to specify these will be discussed in the respective chapters.

3.2.1 Constant definition

Constant classes can be defined to specify constant objects. Constants are defined by means of the def-constant defining form. Its full syntax is:

def-constant ::= (**def-constant** *class-name-spec*
[**:doc** *documentation*]
[**:format** *constant-format-spec*]
[**:methods** {*method-spec*}*])

3.2.2 Primitive definition

Primitive classes can be defined to specify objects which have as far as PHIL is concerned no further substructure. Instances of primitives form leaves of the PHIL world. Primitive classes can be defined in Philgram by means of the def-primitive defining form.

def-primitive ::= (**def-primitive** *class-name-spec primitive-spec*
[**:doc** *documentation*]
[**:format** *primitive-format-spec*]
[**:methods** {*method-spec*}*])
primitive-spec ::= (*type-spec*
[**:construct** *function-spec*]
[**:copy** *function-spec*]
[**:destruct** *function-spec*])

type-spec ::= TYPE-SPECIFIER
function-spec ::= FUNCTION-SPECIFIER

Here type-spec specifies the Common Lisp type of the value of the instances of the primitive class, the :construct option specifies which function to call when an instance of this class is created, the :copy option determines which function to use to copy the value of an instance of this class, when this instance is copied and the :destruct option determines which function to call when an instance of this class is erased from the world. The functions for the options :construct, :copy and :destruct will be applied on the value of the primitive object. The default value for the options :construct, :copy and :destruct argument is the symbol IDENTITY, which specifies the Common Lisp identity function. These functions have to be used to implement the notion of semantical attachment and are discussed in the section "Semantic attachment" of the "PHIL Definers Manual".

3.2.3 Structure definition

Structure classes can be defined to specify objects with a fixed number of sub-objects (records). Structures classes are defined by means of the def-structure form.

def-structure ::= (**def-structure** *class-name-spec structure-spec*
[**:doc** *documentation*]
[**:format** *structure-format-spec*]
[**:methods** {*method-spec*}*])
structure-spec ::= ({*field*}*)
field ::= (*label name* [**:equal** *evaluator-spec*)

label ::= SYMBOL
name ::= *class-name* | *sort-name*

The structure-spec consists of the specification of a list of fields. For each field one has to specify a label, a name and optionally an evaluator-spec. All the labels in the fields of a structure-spec should be unique in the scope of this definition. These labels are used to refer to the components of an instance of a structure class. With name one specifies what sort of sub objects are allowed at that field. The object present at a field in a structure should satisfy the name specified for that field. All names of defined classes and sorts are valid to be used at a name in a field. See also the section "Sort Definition.". The evaluator-spec will be discused in section 3.4.

Fields of structures can also be defined incrementally by means of a def-field form. A def-field form is defined as:

def-field ::= (**def-field** (*class-name label*) *sort-name*)
[**:equal** *evaluator-spec*])

This adds, or redefines the field labeled by label in the class indicated by class-name, which should be a structure class, which has optionally the evaluator evaluator-spec associated with it. Additionally it also defines an acces method to acces object.

3.2.4 Sequence definition

Sequence classes can be defined to specify objects which have an arbitrary number of subobjects (lists). Sequence classes are defined by the def-sequence defining form.

def-sequence ::= (**def-sequence** *class-name-spec sequence-spec*
[**:doc** *documentation*]

[:format *sequence-format-spec*]
[:methods {*method-spec*}*])

sequence-spec ::= (*name* [:minimum *number*])
number ::= (INTEGER 1 *)

In a sequence-spec the name specifies the sort which the components of instances of this class should satisfy. The option :minimium specifies the minimum number of components and is 1 by default.

3.2.5 Sort definition

Sorts specify sets of objects of different classes. The defining form is similar to the defining forms of the classes, except that no format nor method specifications can be supplied. Sorts are defined by means of the def-sort defining form.

def-sort ::= (def-sort *sort-name-spec sort-spec*
[:doc STRING])

sort-name-spec ::= *sort-name* | (*sort-name* {*prop-name prop-value*}*)
sort-name ::= SYMBOL
sort-spec ::= ({*name*}*)

I.e. a sort-spec consists of a list of names. Valid names are all names of the defined classes and sorts. A sort-spec specifies the set of objects which is the union of the sets of objects satifying the names listed.

3.3 Class properties.

PHIL supports a notion of *class properties*. Class properties can be viewed as constant methods. The definer is free to define and use these properties. PHIL itself makes use of two class properties, which thereby are *reserved* properties.

- :FRAME, the value is either **T** or **NIL**. If the :FRAME property is true this means that instances of this class are allowed to be framed. If an object is framed, it means that this object allows textual editing (It must be parsable too). The default value of the :FRAME property is **T**. See section 3.3.1.

- :FILE, the value is either **T** or **NIL**. PHIL contains a notion of *auto filing*. This facility automatically transfers object to file whenever they not needed, and fetches them back when they are needed. The :FILE

property determines whether auto filing may take place for instances of a class. Hence by means of the :FILE property on can control the granularity of objects stored on file. The default value of the :FILE property is **NIL**.

Class properties can be defined as a part of the class definition or separately. The *def-prop* form can be used to specify class properties separately.

def-prop ::= (**def-prop** *name prop-name prop-value*)

where *name* is a name of a class or sort, *prop-name* the key under which the *prop-value* is stored.

3.3.1 Predefined Sorts and Classes

PHIL contains a set of predefined sorts and classes, which thereby are reserved names.

- The class <META>.
  The class <META> is a primitive class. Instances of this class are *templates* or *meta* objects, and can be seen as syntactic meta variables for objects of this class. The value of a *meta* is a *name*. No methods are defined for <META>. *Meta* objects are treated special by the PHIL system.

- The class <FRAME>.
  The class <FRAME> is the primitive class of *frames*. For <FRAME> no methods are defined. The value of a frame is a STRING. Frames allows visual editing and in that are treated special by the system.

- The sort <ANY>.
  The sort <ANY> is the sort of all defined classes and sorts in the system, except <ANY>, <META> and <FRAME>. So whenever a new class or sort is being defined this will be added to the definition of <ANY>. (<ANY> objects are especially useful for the definition of a list of objects of an unknown class, e.g. a kill-list which contains all objects previously killed in the object world.)

In addition to these sorts and classes two more classes are predefined. These are the classes <SYMLAYER> and <SYMSET>. These classes are especially useful in defining evaluators. These classes will be discussed in section 3.4.4.

3.4 The Context Layer

On top of the structure layer lies the context sensitive layer. In this layer, the system invariance is formalized, which incorporates the type correctness as well as context sensitive constraints. A major feature of PHIL environments is that it performs consistency checks after objects have been modified. [1] What checks are executed depends on which evaluators have been defined. Evaluators can be defined for the fields of a structure. By means of an evaluator one can state that a particular object is equal to the object denoted by some expression. Evaluators can only be specified for components of structures.

field :equal (expression)

Evaluators are defined, either when defining a class by providing an optional evaluator-spec, or by a special def-field form. The expression may refer to other objects (a change in these objects will cause the evaluator to be executed), it must create a new object of the same class as field, and it can have some side-effect. (e.g. colouring objects in the object representation.) This will be discussed in subsequent paragraphs.

3.4.1 Referring to other objects

The only way an evaluator (or expression) can refer to another object is to send it a message. If a method corresponding to the received message exists, then this method is executed and the result is returned to the sender of the message.

3.4.1.1 Automatically created methods

For the four existing classes some methods are created automatically :

- for constant and
- for primitive classes one method to access the value of that object is generated. This method is associated with the message :VALUE. Hence sending a message :VALUE to an object O, denoted as (:VALUE O) returns the value of that object, which will be an instance of the Common Lisp type specified for that object.

[1] This is done by incremental attribute evaluation.

- for structure classes one access method for each field will be generated. these methods will be associated with the respective labels of the defined fields.
- for sequence classes only one method is generated. This method will be associated with the message :ALL. Sending the message :ALL to a sequence returns a Common Lisp LIST of all sub objects of that sequence.

One can refer to the object for which a method or evaluator is defined, by means of the variable SELF. The messages provided in this manner are sufficient to acces any descendent of an object. It may however be neccessary to obtain information from other object than from descendant ones. To obtain information from a parent object one can use the RETRIEVE function.

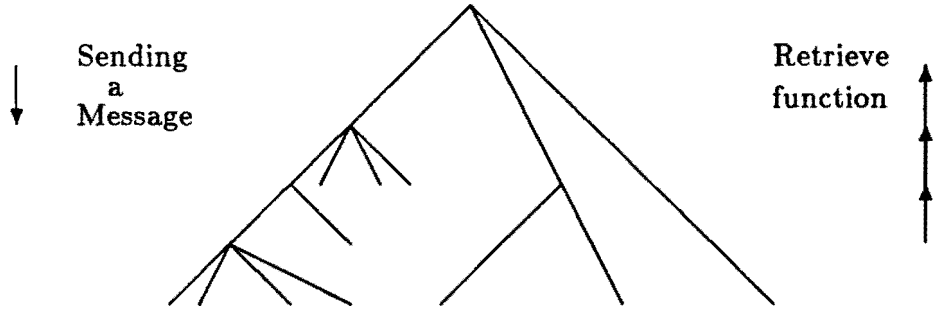

Figure 3.2: Sending Messages through the world.

Function: RETRIEVE (*message &optional (arg T)*)

The function RETRIEVE, which only may be evaluated within an evaluator, behaves as follows. First the message *message*, with the argument, is sent to object in which the evaluator is being executed. If this object does not handle the message, or handles it but returns the value NIL, the message is sent to its parent etc., untill some ancestor is reached which handles the message and returns a non NIL value. This value will then be the value of the RETRIEVE expression. Hence the value of an RETRIEVE expression is dependent on where in the object world the evaluator is executed.

3.4.1.2 Defining methods

In addition to the automatically defined methods, methods can also be defined in a class definition or explicitly by means of the def-method form. When mes-

sages are defined in a class definition, these will be added to the automatically generated methods, or redefine these when a message name is used which is equal to the name one of the automatically generated messages. A method-spec which can be used in a class definition is defined as follows:

method-spec ::= (*message arglist body*)
message ::= KEYWORD
arglist ::= CL-ARGLIST
body ::= {FORM}*

where message is the message for which a method is being defined, arglist the formal parameters of the method, and body the body of the method. FORM is a Common Lisp form with the modified semantics to support message passing. Within the body of a method one can refer to the object the method is executed for by means of the name SELF.

The alternative form to specify a method is a def-method form.

def-method ::= (**def-method** (*class-name message*) *arglist body*)

Where class-name denotes the class for which a method is being defined, and the other elements are the same as in a method-spec. Having both ways available for a definer provides him with the means to modularize the application specification according to his own modularization criteria.

3.4.2 Creating new objects

There are two ways the expression in the evaluator form can create an object. The first possibility is to refer to an object of the same class as the object for which the evaluator was defined. (The object referred to, is copied to the evaluated field.) A new method is presented in this section. Often we would like to specify in an evaluator to denote an object which is not present, but which has to be created. The basic function to create new objects is the function NEW.

Macro: NEW (*class-name &rest value-spec*)

where class-name denotes the class of which an instance should be created, and a value-spec the value, or subobjects of the object. How a value-spec looks like depends on the kind of the class to be instantiated:

- For a constant class the value-spec is empty. So a constant <EQ-OPER> can be created by: (NEW <EQ-OPER>)
- For a primitive class the value-spec is an instance of the type specified for that primitive. So an <INTEGER> can be created by: (NEW <INTEGER> 12345)

- For a structure class the value-spec is a list of alternating labels and objects, specifying the objects at the fields. Fields for which no specification is given, will get a template object. If for a field an object is specified and there is an evaluator associated with that field as well, the object specified by the evaluator prevails.
- For a sequence class a value-spec consists of a list of objects making up the sequence.

3.4.3 Side Effects in Evaluators

Often objects created by evaluators themselves are not shown, but dependent on these objects other objects will be represented in a different way. There are some special side-effects an evaluator can have : colouring of objects and warnings or messages to the user who is working in the environment. Colors are persistent and can only be reset by evaluators. Warnings and messages are given while an evaluator is executed and are therefore lost when more messages are given. It is however possible to associate a message with an object. This message will be displayed whenever the object is selected.

3.4.3.1 Coloring of Objects

In an evaluator one can decide to colour certain objects, i.e. to modify the default representation font. PHIL supports four colours: :BOLD, :REVERSE, :UNDERLINE and :BLINK. Since :BOLD is already used for indicating frames the other three colours can be used to indicate certain states of objects. These colours can be individually set and reset in an evaluator.

Function: SET-COLOR (*object colour*)

Causes *colour* to be set for *object*. The value returned is NIL.

Function: RESET-COLOR (*object colour*)

Causes *colour* to be reset for *object*. The value returned is NIL.

Coloring of objects is in general the preferred way to cause side effects, since they are persistent. This colouring however only works if these objects have a representation on the screen and is therefore mode dependent. If such an object has no representation it can be coloured recursively, which means that all subobjects of this object are coloured. Let's suppose we want to colour an object <O> which is a structure class object, to colour *A* :

```
(def-structure <O> ((sub-obj1 <Sobj1>)
                    (sub-obj2 <Sobj2>)) :format (sub-obj1 (:sp 1) sub-obj2))
```

Coloring object <O> recursively means colouring sub-obj1 and sub-obj2. It is however possible that one of these objects was already coloured in colour *A* because of some error in such a sub-object. Setting of the colour of object <O> to colour *A* (recursively) is oke, but resetting of it may cause some trouble. By resetting object <O> recursively one resets colour *A* of all subobjects of <O> and some error information may get lost. (We loose the error indication colour of one of the subobjects.) There's no problem if different colours are used for colouring object <O> and colouring sub-obj1 or sub-obj2. There are however only three colours available (one is used for framing objects) and this may cause problems, because only three levels can be coloured recursively.

3.4.3.2 Warnings and Messages

In addition to colouring, or instead of, one can also provide messages and or warnings:

Function: MESSAGE (*string &rest args*)

Yields a message at the top of the screen, *string* is a control string as in the Common Lisp FORMAT function and *args* the arguments for it. Its value is NIL.

Function: WARNING (*string &rest args*)

The same as MESSAGE, but also gives an audible signal.

3.4.3.3 Special Messages

Two functions have been defined to associate messages with a node of the phil tree. If such a node is selected by the user the message belonging to this node will be shown automatically in the window. This message is normally used for giving error-information.

Function: SET-MESSAGE (*node colour string &optional recursive*)

Function: RESET-MESSAGE (*node colour &optional recursive*)

These two functions make use of the functions set-color and reset-color. A string will be attached to a node with its colour. Three messages can be stored to a node (object), one for each available colour.

3.4.4 Special Classes

PHIL contains two classes, <SYMLAYER> and <SYMSET> which are very useful in defining evaluators, especially when one is dealing with defined and used occurrences of objects, i.e. where scoped variables are defined. Special functions to create these classes are defined such that a minimal number of recomputations of evaluators takes place after modifications.

3.4.4.1 The class <SYMLAYER>

Objects of class <SYMLAYER>, as well as objects of class <SYMSET>, are not normal abstract structure objects. They serve to facilitate context sensitive checks. <SYMLAYER> class objects are used to store associations between values of objects, and are used in conjunction with the function RETRIEVE. The class <SYMLAYER> is a primitive class. Its value is of type ALIST, a Common Lisp association list. The following messages are supported by the class <SYMLAYER>:

- :VALUE (), which yields the value a ALIST, i.e. the associaton list,
- :ASSOC (*&key test*), yields a CONS, an element of the ALIST or NIL.

The :ASSOC message is similar to the Common Lisp ASSOC function. The test argument, defaulting to the function EQUAL, can be used to provide a test for the keys and the value.

There are two other functions associated with <SYMLAYER>, the function CREATE-SYMLAYER, which creates a <SYMLAYER> object, and MAKE-SYMLAYER which makes a value for a <SYMLAYER> out of an association list of objects.

Function: CREATE-SYMLAYER (*message alist &key key-test value-test*)

This function can only be used in an evaluator to create objects of class <SYMLAYER>. It destructively modifies an old <SYMLAYER> object, according to the alist argument. The message argument must be equal to the name of the message by which things are lookup up from this symlayer. The key-test argument, by default the function EQUAL will be applied on the keys of the alist, and the value-test argument, also by default the function EQUAL, will be applied on values associated with a key.

Function: MAKE-SYMLAYER (*m1 m2 pairlist &key colour key-test*)

The purpose of this function is to make an ALIST, suitable for CREATE-SYMLAYER out of a list of pairs of objects. The resulting ALIST list is an association list, with unique keys, where the keys are the result of sending the message m1 to the first element of the pairs of pairlist, and the value assocaited with it the result of sending the message m2 to the second element of those pairs. A side effect of MAKE-SYMLAYER is that objects, that have the same key value, will be coloured according to colour given. The default colouring is :UNDERLINE.

Using CREATE-SYMLAYER, instead of (NEW <SYMLAYER> alist), is considerably faster, since it only reinvokes those objects which need to be reinvoked because there was a change. This means in the example that after a modifcation of a declaration only those occurrences will be rechecked for which potentially the type has been changed.

3.4.4.2 The class <SYMSET>

Objects of class <SYMSET> are used to denote sets of values. The type-spec for the class <SYMSET> is LIST. Two methods are defined for <SYMSET>:

- :VALUE(), yields a LIST, the value of the object,
- :MEMBER(*value &key test*), a test on membership according the a predicate *test.*

Two special functions are defined with respect to <SYMSET>'s, similar as for <SYMLAYER>'s.

Function: CREATE-SYMSET (*message list &key key-test*)

This function can only be used in evaluators. It desctructively modifies an old <SYMSET> object, according to the *list* argument. The argument *message* must be equal to the name of the message by which things are lookup up from this <SYMSET>. The argument *key-test*, defaulting to the Common Lisp function EQUAL, can be used to specify a predicate to test for equality of values of objects.

Function: MAKE-SYMSET (*m1 list &key colour key-test*)

Similarly to the function MAKE-SYMLAYER there is a function MAKE-SYMSET. This creates from a list of objects a set of values resulting from sending the message m1 to each element if that list. Multiply defined occurrences will be coloured in *colour*, by default :UNDERLINE.

3.5 The Interaction Layer

In the *interaction layer* one defines the user interaction with the objects, i.e. how objects are represented to a user and how a user is able to modify the *object world.* Additionally, facilities to to control the environment itself can be specified.

3.5.1 Interaction model

Objects are represented in a plane. How an object is represented depends on the format specification defined for the classes, and the representation mode defined for the plane. Part of this plane is displayed in a window which is made visible on the screen. Multiple windows and planes are allowed to represent an object in more than one representation mode.

The interaction model in PHIL is based on a command loop. The system receives some events initiated by the user, finds the command associated with that event, executes the command, updates if necessary the representation of the objects in the planes and the windows and waits for the next event to happen. PHIL adopts the notion of a representation invariant, i.e. the

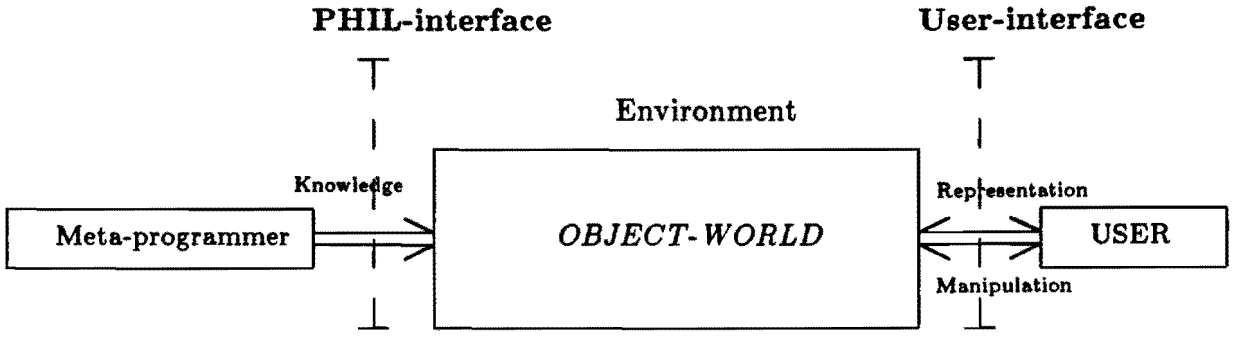

Figure 3.3: Interaction with the *Object World.*

representation of the objects in the various planes is always consistent with the actual objects present and the format specification provided.

3.5.2 Format Specification

In a format specification one specifies how objects are represented to a user, i.e how an object can be translated into a string, which then can be made visible on a screen, on a file etc. This concrete syntax representation is a projection of the internal abstract syntax structures. In PHIL environments it will also be possible to frame certain objects, the idea being that sometimes it is a lot handier to modify the textual representation directly, instead of the object structures. In order to get back objects after representtaions have been edited one needs the possibility to map a string into an object. Normally this process is called parsing. This parser now, for objects which can be framed, has to be specified as well. In PHIL the same format specification, which specifies the representation of objects, also, by default, specifies the parser for those objects. The big advantage is that the printed representation of an object is then by definition always consistent with the parser. The parse strategy used is recursive descent. This causes some restrictions on the parseability. This will be discussed in the section "Parser notes.". Whether or not a parse function needs to be generated depends on the class property :FRAME. If that property is NIL, meaning that objects of that class cannot be framed, there is no need to generate a parse function.

One can have more than one format specification for a class. Therefore format specifications are labeled. The label of a representation will be referred to as a representation mode. When the user wants to view an object he has to choose the representation mode. The representation mode :DEFAULT is special, in that any undefined mode defaults to the :DEFAULT representation mode. A format specification can be supplied either at the class definition, this will then be mode :DEFAULT, or by means of a def-format form where one has to supply the representation mode explicitly. This allows independant specification of structures and projection, and multiple format specification of objects.

def-format ::= (def-format (*class-name mode*) *format-spec*)
class-name ::= SYMBOL
mode ::= KEYWORD
format-spec ::= *constant-format-spec* |
primitive-format-spec |
structure-format-spec |
sequence-format-spec

where *class-name* specifies the name of the class for which one wants to specify a representation, *mode* a keyword specifying the label of the representation mode and *format-spec* the actual format specification. Which alternative is used depends on the kind of the class denoted by the *class-name*. In the following sections the format specification for the various kinds of classes will be discussed.

3.5.2.1 Constant format specification

The format specification for constant classes is the most simple.

constant-format-spec ::= *simple-string* | *case-string*
simple-string ::= STRING
case-string ::= (:case *simple-string*)

If one provides a *simple-string*, that string will be represented. The parser will recognize this string case insensitive, i.e. no distinction is made between upper- and lower-case characters. If one provides a *case-string*, the string will be recognized case sensitive.

3.5.2.2 Primitive format specification

The primitives, together with the constants, form the leaves of the PHIL datastructure. A primitive format specification specifies how primitives are to be represented and how they can be parsed. A primitive format specification consists of three components. A print function, specifying how the value of a primitive object is to printed. This print function, when applied on the value of a primitive object should yield a string, which is used for printing, a parse function specified by a regular expression, specifying how primitives can be recognized (lexical analyzer), and the third element a create function. This create function serves two purposes. Firstly is serves as a filter. For example by means of the create function one can specify that certain strings are not valid identifiers since they belong to a set of reserved words. Secondly the create function actually creates the value for the primitive object. The general form for a primitive format specification is:

primitive-format-spec ::= ([:print *function-spec*]
[:parse *micro-syntax*]
[:create *function-spec*]

Where :print option allows to specify the print function. Its default value is IDENTITY, the :parse option allows to provide the micro-syntax, specifying the lexical structure of the primitive, and is optionally empty. The :create option allows the specification of a filter function on the token obtained after succesful parsing. The default value for this option is the IDENTITY function. For explanation of function-spec and micro-syntax see [1] or appendix A.

3.5.2.3 Structure format specification

The representation of objects of class *structure* is specified by means of a *structure-format-spec*. A structure format specification allows to specify how instances of structure classes are to be represented and parsed. The syntax for a structure specification is given below.

structure-format-spec ::= ({*structure-clause*}*)
structure-clause ::= *character* |
simple-string |
case-string |
object-ref |
spaces |
newlines |
structure-indent |
structure-optional |
structure-field |
structure-format-spec

character	::= CHARACTER
simple-string	::= STRING
case-string	::= (**:case** *simple-string*)
object-ref	::= SYMBOL
spaces	::= (**:sp** *number*)
newlines	::= (**:sp** *number*)
structure-indent	::= (**:indent** {*structure*}+)
structure-optional	::= (**:optional** *object-ref* {*micro-clause*}+)
structure-field	::= (**:field** *length position object-ref*)
position	::= **:left** | **:right** | **:center**

Where:

- A character clause allows the specification of single characters. This will be used mainly to specify non printable characters like a *Newline* character. In the other case the *simple-string* clause will be handier. Note that in the description of the *micro-syntax* also a *character* clause can be specified. There however it is used to specify the parser only. In a *structure-clause* a *character* specifies printing as well as parsing.

- The *simple-string* clause, the same as in the micro syntax for primitives but used for printing as well as parsing.

- The *case-string* clause, for case sensitive recognition.

- The *object-ref* clause allows to refer to the components of the object one is specifying a representation for. An *object-ref* is a SYMBOL, which must be a label in the class definition. the meaning of *object-ref* is that the representation for that component associated with that label will be printed. The labels as they occur in a *structure-format* must be in the same order as they appear in the class definition. Of course not all labels have to be used in the format definition.

- The *spaces* clause allows the specification of a number of spaces between elements of a *structure-format*. Specifying *spaces* results in a *skip white* action of the parser. This means that wherever a *spaces* is specified, arbitrary many (> 1) spaces may occur for the parser to be able to recognize the structure. So frequently the *spaces* clause must be used if one wants this behaviour.

- The *newlines* clause allows the specification of a number of new lines. A newline autiomatically indents to the current indentation level. A *newlines* also results in a *skip white* for the parser.

- A *structure-indent* clause specifies an indention for the *structure-clauses* specified. The indentation level is set at the current print position and all *newlines* will indent automatically to this indentation level. At the end of a *structure-indent* the indentation level is reset to the previous level. Hence complete nesting is allowed.

- The *structure-optional* clause allows the specification of optional representation of part of the structure. The *object-ref* in an *optional-clause* denotes a sub object of a structure. If that object is present the *structure-clauses* in the *optional-clause* will be executed.

- The *structure-field* clause allows the specification of a field of a certain width in which a subobject will be printed. This subobject then should be an instance of a leaf class. If it is not a leaf class , the value of *SHADOW-STRING* (default the string "...") will be used. By means of *position* one can specify where in the field the object should be printed. A *structure-field* always causes a *skip white* before the field and after the field.

3.5.2.4 Sequence format specification

The representation of objects of kind *sequence* is specified by a *sequence-format-spec*.

sequence-format-spec	::= ([**:prologue** *seq-prologue*] [**:indent** *seq-indent*] [**:element** *seq-element*] [**:separator** *seq-separator*] [**:epilogue** *seq-epilogue*]
seq-prologue	::= ({*seq-layout*}*)
seq-indent	::= **T** | **NIL**
seq-element	::= (*number position*) | **NIL**
seq-separator	::= ({*seq-layout*}*)
seq-epilogue	::= ({*seq-layout*}*)
seq-layout	::= *simple-string* | *case-string* | *spaces* | *newlines* |
simple-string	::= STRING
case-string	::= (**:case** *simple-string*)
spaces	::= (**:sp** *number*)
newlines	::= (**:sp** *number*)
position	::= **:left** | **:right** | **:center**

Where:

- The *seq-prologue* clause defines an optional prologue part. The prologue will be "printed" before the contents of the sequence.

- The *seq-indent* clause specifies whether the elements of the sequence should be indented. The indentation starts at the position of the first element, i.e. after the prologue part.

- The *seq-element* clause allows for a field specification of the elements of the sequence. One can specify how long the field should be and where

in the field the element should be printed. When a field specification is applied on a non leaf object a string, which is the value of *SHADOW-STRING*, will be printed instead of the object.

- The *seq-separator* clause allows for the specification of a separator between the elements of a sequence.
- The *seq-epilogue* clause, allows the specification of an epilogue to a sequence. This will be executed outside the indent when the indent option was also specified.

3.5.2.5 Sort format specification

Since sorts are no classes, consequently there are no instances of sorts and no format specification has to be supplied either. However, sorts determine how objects can be linked together to form structures and sequences, so parse functions must be generated for sorts. The parse function generated is directly based on the sort definition. So a definition like for example:

```
(DEF-SORT <STATEMENT> (<CONDITION> <ASSIGNMENT> <WHILE>))
```

leads to a parse function which in order tries to recognize a <CONDITION>, if that fails a <ASSIGNMENT> and if that failes too, a <WHILE> statement. Since the parser tries to recognize the alternatives in the order specified, the order in which classes are listed, is of importance. See also "Parser Notes.".

- Parser notes
  The parser generator is defined such that everything produced by the pretty printer can be parsed. However, also forms not produced by the pretty printer, should also be parsable. If this were not the case the user had to layout the program exactly how it would be pretty printed. Therefore the parser has a notion of *white space*. In all places where the definer specifies either a *spaces* or a *newlines* clause, the parser performs a *skip white space* action. This means that when positioning is unimportant in a particular representation, sufficiently many *spaces* or *newlines* must be provided in the structure and sequence format specifications. White space is considered to be any nonprintable character plus the space character. The parser generator used in PHIL generates recursive descent parsers for the objects. This means that certain restrictions apply on the format specifications for them to be parsable.
- Prefix rule
  In the parse function generated for a sort, the parser tries to recognize the alternatives given. If a particular alternative is succesful, i.e. a string has been found such that it parses correctly in one of the alternatives, that decision will not be revoked, i.e. the parser has a *first fit* strategy.

- Left recursion
  The PHIL parser generator does not support left recursion. Left recursive structures have to be translated into right recursive ones.

3.5.3 Defining Commands

Central in the interaction is the notion of a command. The Phil House Style (See section 3.5.4) provides many standard commands. The environment specifier may define some more user specific commands by using the def-command form :

def-command ::= (**def-command** *command-name arglist body*)
command-name::= SYMBOL
arglist ::= CL-ARGLIST
body ::= {FORM}*

which defines a command for the symbol denoted by *name*. *arglist* is the arguments list for the command. Since commands are mainly executed at *top level*, i.e. directly user initiated, there are certain constraints on *arglist*.

- The required arguments are used by the command former, i.e. are promted for by the system when not supplied. See the user interaction document.
- All other arguments should be defined as *&optional*, *&key* or *&rest* arguments, as allowed in a CL-ARGLIST, with suitable defaults.

The *body* is the same as the body as allowed in a function definition and consists of a sequence of FORM, i.e. a Common Lisp form. Within the body one can refer to the current plane by means of the variable *PLANE*. This variable returns the a datastructure of type PLANE, and provides access to plane dependent information like the *plane focus*, the *representation mode* etc. The datastructure plane, how to create them and what functions are defined on planes, are defined in the user interaction document.

3.5.4 The Phil House Style

This section describes the set of interaction facilities, to be referred to as the Phil House Style (PHS), for environments generated by means of the PHIL environment generator. The PHS is delivered with the PHIL generator and is incorporated in the helix-environment. The PHIL interaction style is described in [6] and [2]. Some extra commands are defined in section 6.5.

Interaction with the helix-world is event-driven: generated events, like pressing a button, hitting a key on the keyboard invoke associated *commands*. PHIL events are generated by *devices*.

A PHIL window is a sunview frame and can contain at most one *main* pane in which a PHIL object is represented. The other panes can be *panels*, containing *buttons, sliders, toggles etc.*, and *textsw's* general purposes text windows. Panels allow the specification of seven kinds of devices to be present in the panel:

- **message** devices, text strings that can serve as annotations or expose dynamic status messages
- **button** devices, that allow direct command initiation
- **choice** devices, that allow to select one choice from a list
- **cycle** devices, that are essentially choices with a different appearance
- **toggle** devices, that allow for the selection of a number of choices from a list
- **text** devices, that provide type-in fields
- **slider** devices, that allow graphical value setting

How these devices are defined for a panel is explained in [4]. The used device specifications are given:

The *device-specs* for the different devices is treated in the next sections. The common keywords in the specs denote:

```
:label      The device label string
:row        The sunview panel row at which to place the device
:col        The sunview panel column (zero based)
:left       The associated command with the left mouse button.
:middle     The associated command with the middle mouse button.
:right      The associated command with the right mouse button.
```

All keyword arguments, except *:row*, are optional. *:label* defaults to the empty label string, except for buttons, where the button identifier will be used as the label. *:col* defaults to the column next to the previous device.

3.5.4.1 Messages

A message *device-specification* has the form:

```
(:message &key label row col left middle right)
```

3.5.4.2 Buttons

A button *device-specification* has the form:

```
(:button &key label row col length font left middle right)

:length  The width of the button in columns (default adapted to label)
:font    The label string font (default the panel font)
```

3.5.4.3 Text

A text device-specification has the form:

```
(:text &key label left middle right row col)
```

3.5.4.4 Text windows

Text windows are defined as:

```
(:textsw &key contents)
```

The contents key allows the specification of an initial contents of the text window.

3.5.5 Accessing panes and devices

The following variables are always available during interaction with the PHIL system and can be referred to by commands:

- *plane*, the current plane
- *pane*, pane the current pane
- *device*, device the current device

For a complete set of accessors see the PHIL manual. The most important accessors are:

```
kb-focus (&optional plane)        => object, the current object
kb-top(&optional plane)           => object, the top object
plane-pane (pane-name)            => pane, the indicated pane
get-device (device-name)          => device, the indicated device
set-device-label (device string) => to set the label of a device
```

```
get-device-label (device)          => string to get the device
                                      label string
set-device-value (device value)    => to set the value of a device
get-device-value (device)          => value to get the value of a device
```

A hardcopy of the default PHIL plane, as defined in [4], is given by appendix K.

A complete list of PHS commands is given in [2]. Interaction with the Helix environment is fully described in [6].

Chapter 4

The Helix Object Layer

The logical structure of the Helix-environment is described in this section. All the objects allowed in the helix-environment are given by the BNF-syntax specification of HHDL given in APPENDIX A of the [5]. This specification can be translated directly in Philgram. There are however some modifications made :

- ☐ All left recursive structures are transformed into right recursive structures
- ☐ Objects in sort specifications are ordered according to the prefix rule
- ☐ Sequences must of be length one or more
- ☐ Some new objects are introduced

This Philgram specification can be found in [7]. From this Philgram specification we can generate a BNF like syntax form by using the function GENERATE-SYNTAX [3], which is given in appendix B. This syntax-specification should be the same as the original one. The modifications are given in the following sections.

4.1 A Top Level definition.

If we want the user to be able to create more than one module or package in the helix-environment, we have to define a so called top-level. The top-level object <hhdl_programs> is defined to be a sequence of modules and/or packages.

4.2 Standard Routines.

HHDL contains some standard functions and procedures that can be used by the model-describer. There are two ways to introduce standard functions and

procedures in the environment. The first possibility is to define a constant symlayer object for the top node in the helix-world which contains all the necessary information. Functions and procedures are stored in the same way user-defined routines are. A disadvantage is that the names of those routines are not available for the user. The user may use these standard routines and they are checked all right, but a misspelling of a routine name e.g. results in an error indication because the system doesn't recognize it. A much nicer way is to define the structures of these routines and add the set of these routines to a known sort-specification. The user may then select a needed predefined function or procedure from a menu, given by the environment. We define the sorts <predefined-procedure> and <standard-function> to be the sets of predefined procedures and functions respectively. The sort <predefined-procedure> can be added to the elements of the sort <unlabelled-statement> and <standard-function> to the sort <factor>.

4.3 Predefined Packages.

Part of the Helix-System are some predefined packages written by SILVAR & LISCO to support the user. The specification for <use-part> is therefore changed to make the predefined package names <Reg_Pack>, <BitPack>, <LogPack>, <Utl_Pack>, <Dr_Pack>, <Rnd_Pack>, <StrPack> and <IoPack> available. (see [7]). More than 150 functions, procedures and types are defined in these packages. Correct usage of this information must be checked and the knowledge we have about these predefined packages must therefore be stored in the system. The easiest way i.e. the fastest way to do this is to define a constant symlayer object which contains the predefined data in some way (see section 5.7). A problem however is, that a complete list of functions and procedures is not available for the user of the environment. The remedy for this is the presence of an on-line manual (see section 6.5.1).

4.4 Ordinal Types.

Some standard HHDL-types must be available for the user and are not directly given by the syntax specification of HHDL in [5]. The types <boolean-type>, <char-type>, <real-type>, <integer-type>, <semaphore-type> and <text-type> can join the sort <simple-type>.

4.5 Comment in the helix environment.

It often happens that models described in HHDL are not easily understood. Thus, it is helpful and good programming practice to annotate the source-code by inserting explanatory comment-strings. A comment-string is a string starting with '(*' followed by some characters and ending with '*)'. The HHDL syntax-checker simply ignores these strings and they are thus allowed in all places in the code. In the environment there is a notion of structure : objects are only allowed in defined places. One way to allow comment in the helix-world is to define an optional comment-object for every place possible in all defined structures and sequences. A more convenient way is chosen by the Phil generator designer. Every node in the world is given a so called *comment*-property which results in the possibility to attach a string to a node in the tree. The string will be displayed if this node is selected with the mouse.

4.6 Unlabelled Statements.

The context free structure of hhdl is given by appendix A of [5]. The structure of the sort <unlabelled_statement> given by this appendix must be modified. The <detach_stat> is only allowed in the model descriptions (comptype/translator) and is therefore removed from the original set given. A <comment>, <empty> and <predefined-procedure> is added to the elements of the sort-specification. The *empty* object is the string "(*empty*)" and not the empty string "", because empty strings give parsing problems. The *comment* object is the only comment that is directly visible in the helix-tree representation.

A new kind of statement is the <model-unlabelled-stat>. For modelling, some extra statements are available. A <model-unlabelled-stat> is a normal <unlabelled-statement> completed with the statements <sensitize-stat>, <desensitize-stat> and <detach-stat>.

In subprocesses some extra statements are available. We define the <subp-statement> to be the sort <unlabelled-statement> completed with the statements <wait-stat> and <signal-stat>.

4.7 Redefining some Sorts.

The sort-specification of <constant> and <factor> given by [5] can be modified slightly because some predefinitions are available for the user. The primitive <char> and the constants <true> and <false> may be added to both sort

specifications. Furthermore the set of standard functions <standard-function> may be added to the sort <factor>.

4.8 Procedure and Function declaration.

The structure definition for <proc_func_decl_part> given by appendix A of [5] is not correct. The specifications given indicate that this object is a sort specifier specifying only *one* function or procedure. This musty however be a sequence of functions and procedures. There are four kind of routines : *normal*, *forward*, *extern* and *fortran*. To simplify context-checking the <proc_func_decl_part> is chosen to be the list of the set <proc_func>. The latter is a set of 8 kinds of routines : <procedure>, <function>, <forward_procedure>, <forward_function>, <extern_procedure>, <extern_function>, <fortran_procedure> and <fortran_function>.

4.9 BNF Transformations.

At the time this application was written, left recursive structures were not allowed in Philgram. That's why left recursive structures are transformed into right recursive structures. The structures of <variable> and <simple_exp>, given by [5] are transformed into the structures given by appendix B.

4.10 Parser Optimizations

Consider for example the following definitions

```
(def-sort <expression> (<rel_exp> <simple_exp>))

(def-structure <rel_exp> ((simple-exp1 <simple_exp>)
                          (rel-oper <rel_operator>)
                          (simple-exp2 <simple_exp>))
               :format (simple-exp1 rel-oper simple-exp2))
```

The parser function generated for the sort <expression> first tries to recognize a <rel_exp>, i.e. a simple_exp followed by a rel_oper and a simple_exp. If this is not succesful it tries to recognize <simple_exp>, so in that case the first <simple_exp> is parsed twice. A parser optimization form can be defined to avoid double parsing of text.

```
(def-parser (<expression> :default)
```

```
            (alternatives <rel_exp>))

(def-parser (<rel_exp> :default)
            (right-recursive (simple-exp1 <simple_exp>)
                             (rel-oper <rel_operator>)
                             (simple-exp2 <simple_exp>)))
```

This causes the simple-exp1 to be returned if parsing of the <rel_exp> is not succesful. An optimaization form can be defined for the sorts : <expression>, <simple_expression>, <unsigned_simple_exp> and <element>. These parser-optimizations can be found in [7].

Chapter 5

The Helix Context Layer

On top of the Object Layer a Context Layer must be defined. In this layer we define the invariant relations that must be present in the helix world. The environment performs consistency checks e.g. type-checks, after objects have been modified. What checks are executed depends on which evaluators have been defined.

Most of the checks we want the environment to perform are type checks. The type of an object normally depends on declaration information. What we need is a mechanism to retrieve this information. Evaluators can get information from an other objects by sending a message to it. Only those objects for which a corresponding method was created, respond to this message . Normal message sending is used to obtain descendant information, the retrieve-function message sending to obtain ascendant data. Objects of the special class symlayer and symset can be used to simplify data retrieval. See [7] for the actual implementation of these symlayers.

5.1 Performing Checks using Evaluators

Normally checks are performed by defining evaluators. Additional fields with an evaluator-specification are defined for structures by using the def-field form. Evaluator forms have some side-effects which depend on information found in the helix world. Normally an evaluator-field object has no representation on the screen and may therefore be called an internal object. The standard evaluator specification is shown :

(def-field (*class-name1 label*) *class-name2*
:equal *(new class-name2 (:message self args))*)

(def-method (*class-name1 :message*) *(args)*
(*retrieve* information by sending messages and retrieve function)
(give user a *message*)
(*colour* (sub)objects of *self*)
(return *new* object of class class-name2))

An object of class *class-name2* is defined for the structure object *class-name1.* In the evaluator-specification a new object of class *class-name2* is created by means of the function **new**. A message *:message* is sent to the object of *class-name1* to obtain an appropriate form to specify objects of *class-name2.* The method *:message* for objects of class *class-name1* will respond to the received message. Side effects are given by the method-specification. The object of *class-name2* could be defined as a special primitive object which cannot be framed and will be, for debugging purposes, shown on the screen as a string. The Lisp type of this object is always correct, indicated by **T**.

(def-primitive (<s-exp> :frame nil) (T)
:format (:print princ-to-string))

Let's suppose object O must be checked for some reason. (figure 5.1). An evaluator-field of class EO is defined for the structure object O. The method, called *:message*, defined for this object must respond to the message sent by the evaluator. It will decide if something is wrong with an object and will colour objects if this is the case.

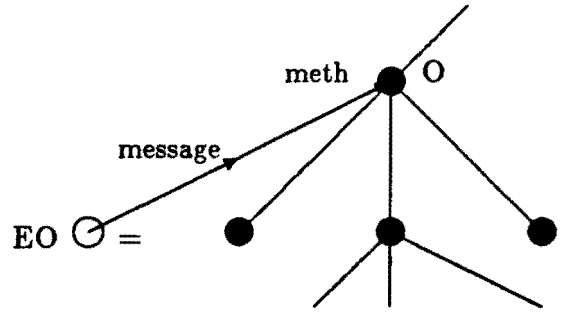

Figure 5.1: Defining Evaluators.

There are four kinds of checks we want to execute :

- **type specification checking.**
  By defining the structure (or better the sort) <TYPE> we globally define what kind of types exist. In our design environment however not all type specifications are allowed. The type specifications that are allowed can be found in appendix C.

- **type checking.**
  A major task for the environment is to check types. If incorrect types are used, the system has to notify the user in some way. Objects can be coloured or warnings can be given to the user.

- **double definitions checking.**
  Normally double definitions in declaration parts are not allowed. By choosing apropriate keyvalues in symlayers one can make use of the convenient side effect of creating symlayers : colouring of double occurences. For other double definition checks one can define extra symlayer or symset objects or check these by means of a Lisp function.

5.2 Type Specification Checking

Type specifications are normally given by a <TYPE> object. The global structure of instances of this object is given in the object layer of the specification (appendix B). More constraints on type specifications are given by [5] and are listed in appendix C. Evaluators have to be defined in the context layer to check if instances of <TYPE> satisfy the given constraints (see appendix I).

5.3 Type Checking

The helix environment must eliminate the possibillity to make certain errors and warn the user about inconsistencies in the source program, for example type conflicts. Fast feedback must be given by the environment about the validity of modifications made by the user in the source program. A complete list of type checks that have to be done in the environment is given in appendix I. By defining the **Helix Object Layer** the global **HHDL** structure is known by the environment and can be manipulated by the user. For example the following statements may be used in HHDL:

IF <EXPRESSION> **THEN** <STATEMENT>

<VARIABLE> := <EXPRESSION>

The *expression* object is used in two different contexts. In the *if-then* statement the experession must be of type BOOLEAN, whereas in the *assign-statement* its type must be equal to the variable-type. What we see is that for all the different contexts (context objects CO in figure 5.2) an evaluator-field object (EO) must be defined which checks the type-relation between objects.

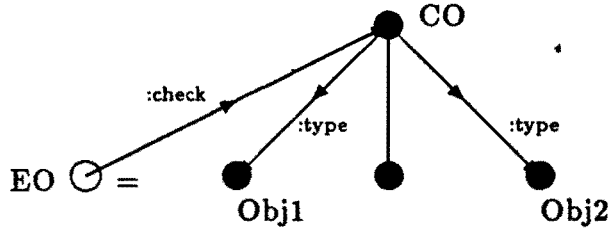

Figure 5.2: The Type Evaluator.

For all objects that are of a certain type a *:type*-method must be defined which returns the object-type. These objects are listed in appendix D.

We now introduce three kinds of messages an object of the classes listed in appendix D should respond to. These are : **:ctype, :vtype** and **:type**. The methods responding to these messages are defined in [7]

- **The Method :ctype**
  The :ctype returns the type of an object if it is of kind constant. Methods have to be created for the objects of classes appendix E.

- **The Method :vtype**
  The method :vtype returns the type of an object if this object is of kind variable. Methods have to be created for the classes mentioned in appendix F.

- **The Method :type**
  The method :type returns the type of an object if this object is of kind variable or constant. Methods have to be created for the objects of the classes mentioned in appendix C

The types of some objects cannot be determined directly because they depend on declaration information. This information must be retrieved in some way from higher up in the helix-tree. How this is done is explained in section 5.4.

5.3.1 Determining the type of a HHDL variable.

A HHDL variable is defined by:

<VARIABLE> ::= <IDENT> <SELECTORS>

where <IDENT> is an identifier and <SELECTORS> is a sequence of <VARIABLE_SELECTOR>s. A <VARIABLE_SELECTOR> is defined as the sort specifying the classes <POINTER_DEREFERENCE>, <FIELD_SELECTOR>, <ARRAY_INDICATOR> and <SUBRANGE_INDICATOR>.
If the first selector is a pointer-dereference then the identifier must be of pointer-type. The remaining type (the type pointed to) is used for further selection using the remaining selectors.
A field-selector is used to select the field of a record variable. The type of this field is used for further selection.
Selection of an array component is a bit more complicated because arrays of arrays are allowed too.
An example-array **ARRAY**[1..10,2..4] **OF ARRAY**[3..6] **OF INTEGER** has the same dimensions as **ARRAY**[1..10,2..4,3..6] **OF INTEGER**. Components of this array can be selected in many different ways. The component [1][3][4] is the same component as [1,3,4] or [1][3,4]. The selection mechanism for array components is therefore a bit more complicated than for other selectors. This mechanism will be described.

Let's suppose a variable is given by:

var : **ARRAY**^{i} **OF** component-type$_i$

This is an *i-dimensional* array (ranges given by i simple-types) of a certain component-type *component-type*$_i$. This component type however can be an array-type, say

component-type$_i$ = **ARRAY**^{j} **OF** component-type$_j$

The new array constructed from these two arrays is equal to the original specification

var : **ARRAY**$^{i+j}$ **OF** component-type$_j$

which describes an i+j dimensional array (ranges given by i+j simple-types).

The new component-type$_j$ can be an array again so this procedure is executed for all component-types and will result in a construction with a non-array component-type. The final array construction is used in the selection procedure to obtain an array-component. The dimensions of the array-selector must be less than the dimensions of the constructed array. If the dimensions of an array-selector are less than the dimensions of the constructed array-dimensions, part of the array will be used with the remaining selectors. The array-selector-dimensions may not be greater than the constructed array dimensions. The implementation of this mechanism can be found in [7].
A subrange-indicator is used to select a subrange of a hhdl register-variable.

5.4 Storing Declarations using Symlayers

There are two ways to retrieve declaration information. Both ways make use of the retrieve function because 'parent' information is needed. The first possibility is to send a message up the tree (see figure 5.3). This message is received by the declaration node (object O_d) and the method belonging to this message will send another message down the tree to obtain the needed information (i).

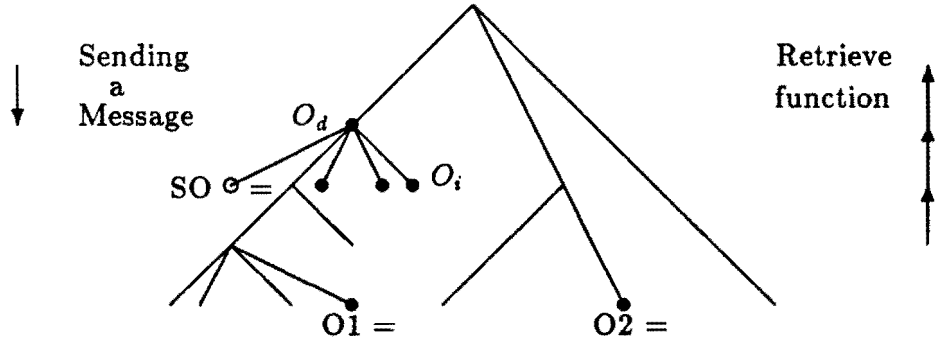

Figure 5.3: Retrieving declaration information.

This can however be done in two steps. First an extra field of class symlayer (SO) is defined for the declaration node(O_d). An evaluator for this field (=) will take care that this object will always contain updated declaration information (i). A convenient side-effect of using this symlayer is that double defined occurrences are coloured.

(def-field (<O_d> :syml) <SYMLAYER>
:equal (create-symlayer :lookup-type

```
                                    (make-symlayer
                                        :keyvalue
                                        :data
                                        (make-assoc-object-list))))

(def-method ( <O_d> :lookup-type) (id)
            (:assoc (:syml self) id))
```

The *make-assoc-object-list* function must create an object pairlist. The actual symlayer information will be the result of sending the messages *:keyvalue* and *:data* to the first and second objects respectively of the alist-elements. The object-alist will be something like :

$$(\{(\ key\ data\ )\}^*)$$

A *:keyvalue* method and a *:data* method must be defined for the key-objects and the data-objects respectively. By defining the method *:lookup-type* for the class $<O_d>$ we are able to obtain information from the defined symlayer. Again, information is looked up by sending the message *:lookup-type* up the tree using the retrieve function.

5.4.1 Determining Scopes

Useful objects to store declaration information in, are objects of class $<SYMLAYER>$. This information can then be retrieved by children of the node for which this symlayer object was defined. The structures for which an extra field of class symlayer must be defined, depend on the scope of the declaration. O1 for instance in figure 5.3 belongs to the scope of O_d, whereas the evaluator for O2 will not be able to obtain information from O_d. The structures for which a symlayer field must be defined and the information that has to be stored in that object are given by [5]. See table 5.1. Scopes are given horizontally, symlayer contents vertically.

The information we want to store consists of two parts : a key or keyvalue and some data. The keyvalue is normally an identifier by which we look up the data. The data is the information we would like to have on this identifier.

(keyvalue data)

The keys and the data we store in the defined symlayers is given in the following

	COMPTYPE TRANSLATOR	MODULE	PACKAGE	PROCEDURE FUNCTION	HHDL-PROGRAMS
pins	yes				
labels	yes		yes	yes	
constants	yes	yes	yes	yes	
types	yes	yes	yes	yes	
variables	yes		yes	yes	
enum const	yes	yes	yes	yes	
ct pars	yes				
nettypes		yes	yes		
clocks		yes			
fp		yes	yes	yes	
fp pars				yes	
packages		yes	yes		
global					yes

Table 5.1: Scopes and symlayers contents.

sections.

5.4.1.1 Pins

Components are connected to each other by use of connections between pins. These pins must be defined in the <net_ref_part> of the component type specification. There are four kinds (*directions*) of pins : INWARD, OUTWARD, BOTHWAYS or INTERNAL. The type of a pin is given by the nettype-descriptor which must be defined by the user in the <nettype_decl_part>. The information we store is:

(identifier 'PIN direction nettype)

5.4.1.2 Labels

There are two kinds of labels : declared labels and actual labels. An actual label is a label that is used in labelled statement. Such a label must however be declared in the <label_decl_part> before it is used. Actual labels can be jumped to by using a <goto_stat>. The information that is stored to enable label-checking is:

((identifier 'DECLARED) 'LABEL)

((identifier 'ACTUAL) 'LABEL)

The key under which information is looked up is not a simple string (the value of the identifier), but a list containing two items : the identifier-string and the symbol 'declared or 'actual. These keys are chosen to realize proper double definition checking. A declared and an actual label with the same integer value are allowed now.

5.4.1.3 Constants

Constants, of integer, real, character or string type must be defined before they are used. The information we store is:

(identifier 'CONSTANT constant-type-spec-list)

where constant-type-spec-list is a list containing type information, determined by sending the message *:type* to the constant in the <const_def> of the <const_def_part>.

5.4.1.4 Types

One can define an identifier to be a certain type. It's much easier to use symbolic names for type specifier. Types are specified in the <type_def_part> of the declarations. The information stored is:

(identifier 'TYPE type-spec-list)

The type-spec-list is determined by sending the message :ptype to the <TYPE> object in the type definition. This message :ptype is explained in section 5.4.2.

5.4.1.5 Variables

Variables are symbolic memory places. Variables may be defined by the user in the <var_decl_part> of the declarations. The information we store is:

(identifier 'VAR type-spec-list)

The type-spec-list is determined by sending the message :ptype to <type> in the <var_decl_part>.

5.4.1.6 Enum Const

(Enumerated Constants) If an enumerated type is defined then the enumerated elements of this type may be stored as constants.

(identifier 'CONSTANT enum-type-spec 'ENUM)

The *enum-type-spec* is determined by sending the message *:ptype* to the <type> specification. Instead of the symbol 'ENUM one could store information about the object in which the enumerated constant was defined.

5.4.1.7 CT pars

Comptype and Translator parameters are stored in a local symlayer, because they are local variables for these objects.

(identifier 'VAR type-spec 'PARM)

The symbol 'PARM is used for checking of the default assignments in the <default_decl> of the comptype objects.

5.4.1.8 Nettypes

Pins must be of a certain nettype. Nettypes are defined in the < nettype_decl_part> of the declarations. The information stored is:

(identifier 'NETTYPE nettype-spec-list)

The nettype-spec-list is again determined by sending the message :ptype to the <TYPE> object in the nettype-decl-part.

5.4.1.9 Clocks

For synchronisation purposes clocks can be used. A clock however must be declared before it is used. That's why we store it in a symlayer object. Clocks are defined in the <clock_def_part>. The only information we have got to store are the names of the clocks:

(identifier 'CLOCK)

5.4.1.10 FP

(Functions and Procedures) Functions and procedures must be declared before they are used:

(identifier pf-kind result-type-list indication parms)

where pf-kind is 'FUNCTION or 'PROCEDURE
result-type-list is the result-type of a function
indication is 'NORMAL for a normal function declaration
'FORWARD for a forward declaration
'EXTERN for an extern function or procedure
'FORTRAN for a fortran function
parms is a list of parameter types, kinds and names
the parameter kind can be 'XVAR for VAR parameters
and 'VAR for other parameters

5.4.1.11 FP pars

(Function and Procedure parameters) Parameters are local variables in the block-part of a function or procedure and are therefore stored in a local sym-layer. The information stored is:

(identifier 'VAR type-spec 'PARM)

5.4.1.12 Packages

Packages normally contain functions and procedures for the helix-programmer to use. If some function or procedure is used in e.g. a module, the package name in which this function or procedure is defined, must be added to the <use_part> of this module. The information we store is:

(identifier 'USED)

5.4.1.13 Global

Modules can make use of function, procedure, variable and type-definitions from defined or predefined packages by means of the use-statement. Symlayers have been defined for all programs (modules and packages) in the helix-world. An object in one package or module can only obtain information from another package through the parent-node of all programs. For this node (or object) a symlayer of symlayers is defined containing all the information available in the world.

(package-name symlayer)

Information from other packages can now be accessed by retrieving a complete symlayer.

5.4.2 The Method :ptype

Important information that is stored in symlayer-objects are type-specifications. To obtain the type information of a <TYPE> object, we simply define a *:ptype* (permitted type specification) method for all objects of the sort <TYPE>. This method must return the type information of this type as a list. The *:ptype* methods can be found in [7] and appendix C.

5.5 Double Definitions Checking

Double definition of an object is mostly forbidden and always confusing. That's why double definitions must be detected in our helix environment. Double appearances are checked for automatically when using objects of type symlayer or symset. Double definition checks are given in appendix I. The standard construction for creating a symset is

```
(def-field (<O> :syms) <SYMSET>
        :equal (create-symset :defined
                              (make-symset
                                  :value
                                  :colour colour
                                  (make-obj-list))))
```

The information in this symset-object is not needed by any helix-object, so no method *:defined* is created to allow data-retrieval from this object. A *:value* method must be defined for the objects assembled by the *make-obj-list*-function. Objects that have the same *:value* result are coloured according the colour-key *colour*. The colouring by symset-evaluators may not be in conflict with colouring by other evaluators. The colour of an object may only be set and/or reset by one specific evaluator.

5.6 Interpackage Communication

The HHDL package file is used to define subroutines which can be referenced by the models of any HHDL module file. The main purpose of packages is to give the user a modular capability of extending the HHDL language.

Packages can only be referred to if they are in use, i.e. if they are present in the <USE_PART> of the <MODBLOCK_PART> or <MBLOCK_PART>. One way to create the retrieval mechanism for used packages will be explained.

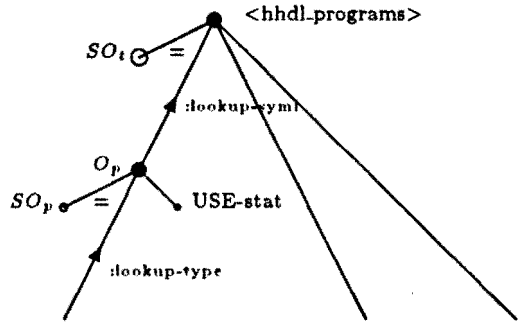

Figure 5.4: Using other packages.

First a symlayer object SO_t (see figure 5.4) is defined for the top-object <hhdl_programs> in our Helix-world. This object contains the symlayers of all packages present in the world. A symlayer can be retrieved by sending the message *:lookup-syml* with argument *pack-name*, up the tree. The complete symlayer with that name will, if it exists, be returned. Secondly an object of class symlayer SO_p is defined for the first parent structure-objects of <USE_PART> (<MODBLOCK_PART> and <MBLOCK_PART>). These objects contain symlayer information of those symlayers that are in use i.e the ones that are listed in the *USE-statement*. This information is looked up by sending the message *:lookup-syml* with argument a package name, up the tree. Let's suppose the message *:lookup-type* is used to retrieve declaration information. The only thing we have got to do is to define a method *:lookup-type* for O_p which tries to look up data in SO_p. The implementation of this mechanism can be found in [7].

5.7 Using Predefined Packages

A number of standard packages are available to supplement the basic HHDL language in following areas :

- **Reg_Pack** : This package provides register manipulation capablity which allows the user to model arbitrary length registers. Arithmetic and l ogical operations on registers are supported.
- **BitPack** : This package provides bit string manipulation capability which allow the user to query and set bits of 32-bit integers. Bitwise arithmetic and logical operations are supported.
- **LogPack** : This package provides the user with a set of routines that simplify logic-level modelling. The routines assist in such tasks as logic-level-to boolean conversion, delay handling, and integer-to-bit conversion.
- **Utl_Pack** : This package provides the user with a set of general support routines for performing tasks such as returning circuit information (number of nets, number of components, etc), controlling trace outputs and others.
- **Dr_Pack** : This package provides a number of routines to facilitate restoring values of local comptype variables when a simulation restart is performed.
- **Rnd_Pack** : This package provides a number of random number generation subroutines for generating various forms of random numbers.
- **StrPack** : This package provides the user with a set of routines for performing string manipulation. The routines assist in tasks such as insertion, concatenation and searching.
- **IoPack** : This package provides the user with a set of routines for performing input and output tasks. The routines assist in tasks such as lexical analysis and parsing.

The information stored in these packages can be added easily to the information already present in the Helix world. A constant symlayer objects can be defined for the top object <HHDL_PROGRAMS>. The mechanism already present for used packages (see section 5.7) takes care of appropriate retrieval of the information that is stored in these symlayer objects. A constant symlayer is created by means of the function NEW because no objects are available to be used with the functions **make-** and **create-symlayer**. No double definitions are checked but this is not needed anyway. Information is again stored in a way all symlayer information is stored. A global Lisp variable ***predefined-packages*** can be used to store the predefinitions in. The redefinition of the method *:lookup-syml* takes care of the predefined package information retrieval. The implementation can be found in [7].

5.8 Special Symlayers

In this section some special symlayer definitions are described. The specifications in lisp can be found in [7].

5.8.1 Forward Symlayers

It is sometimes necessary for two routines to mutually refer to each other. It is also sometimes desirable, in order to make a program more readable, to list routine declarations in one place, and the body of the routine code elsewhere. These situations can be handled by use of FORWARD declarations.

If a routine is forward declared then the parameters and the type-specification, (for functions) is given in the forward declarations and not in the corresponding "normal" declaration of the routine. Parameters will normally be referred to in the body of the "normal" routine, so we need a special mechanism to access the data defined in a forward declaration.

To realize the retrieval of the forward declared parameters and the type-specifier a symlayer object *forward-syml* is defined for <function>s and <procedures>s, which contains all parameters. These parameters are stored in the symlayer as if they were normal variables. A defined method *:lookup-type* for the function and procedure routines will now try to look up identifiers in this special symlayer.

5.8.2 Assignment to Function variables

In the assignment statement a value of an expression is assigned to a (list of) variable(s). The type of the variable, determined by sending the message *:vtype* to the <variable>-object, and the type of the expression, determined by sending the message *:type* to the <expression>-object, must be equal. Within the block of a function, a value may be assigned to the function-name variable. Function-names are stored in a symlayer as functions and not as variables, so the normal *:vtype* method can not be used to retrieve the type of a function-name variable.

This problem is solved by defining a symlayer object *funcvar* for the <function>-object. The contents of this object are determined by looking up the type of the function self or the type of its corresponding forward version. A method *:funcvartype* for <variable>-objects can be used retrieve the type of a function-name variable.

5.9 Phil Redefinitions

The Phil Environment Generator is not a commercial product and is still subject to modification. Some modifications have been made to simplify working with the generator. See [7].

5.9.1 Redefining the function (RE)SET-COLOR

Evaluators can make use of the side-effect of colouring objects. The functions SET-COLOR and RESET-COLOR can be used in two modes : normal and recursive. In the normal mode the colour of an object can be set or reset. However, if this object has no representation on the screen the user won't notice the effect. One way to prevent this is to colour objects recursively, i.e. all the subobjects of the object are coloured also. This may cause problems. If an object is coloured recursively, its colour must be reset recursively too. This means resetting that colour in the whole subtree of the object. This could mean loss of information because such a subobject could have been coloured in the same colour for some reason. What we would like to do is to make use of multi-level-colouring :

To do this every node is given four node properties, :bold, :reverse, :underline and :blink. These properties are colour-counters. We now colour objects always recursively and we don't colour meta-objects. Setting the colour of an object means incrementing the colour-counter, resetting means decrementing it. Properties are introduced or removed when necessary. Eventually a helix program will be correct and all these properties are thereby removed. Functions are used from [3] to solve the problem. Redefinitions of the function SET and RESET-COLOR can be found in [7].

5.9.2 Redefining the macro NEW

Redefinition of this macro is the result of a change in the internal structure of PHIL. Explanation of it would go beyond the scope of this report.

5.9.3 Redefining the macro DEF-CONSTANT

The new macro is a copy of the definition of DEF-CONSTANT from an earlier version of the generator PHIL. This definition adds the automatically created method :value to constant-definitions. The redefinition is again an internal problem and will not be explained any further.

5.10 Phil Bugs

Some internal Phil problems are not solved yet :

5.10.1 Auto Frame

Phil permits editing of text as well as editing of objects. An *auto-frame* function immediately removes the selected meta-object if the user tries to type in some text on his keyboard. This text is automatically framed and must be parsed by pressing the "line-feed" key or by using the parse-command function (selecting the "frame" button with the right key on the mouse). The *auto-frame* function however does not work properly. If a meta object is selected and some text is typed on the keyboard, the text is sometimes placed in the object to the right of the selected object. This is an internal Phil problem and can not be repaired by the author.

5.10.2 Eval

One of the commands defined in the standard Phil interaction interface (PHS), is the "Eval" function which evaluates the expression that is given in the text-pane. A minor problem is that the machine crashes if the text-string is empty ("").

5.10.3 Retrieve

In HHDL, it is possible to declare FORWARD functions and procedures. We need to check if a forward routines precedes the actual declaration of the routine body. A function *RETRIEVE* can be used in PHIL, to look up information in part of the object-tree. What we would like to have is two more functions: *FORWARD-RETRIEVE* and *BACKWARD-RETRIEVE* to look up information in the right and the left part of the object-tree.

Chapter 6

The Helix Interaction Layer

In the Helix Interaction Layer the user-interface is defined. A standard interface called the Phil House Style (PHS) can be used to interact with the generated environments on a SUN workstation. This standard interaction facility incorporates many commands to manipulate with objects and text.
The helix object-tree, or a subobject of it, will be represented in a SUNVIEW plane and a part of it is visible in a window. An example is given in appendix K which visualizes the default interaction plane. The representation of objects depend on the mode that is selected by means of the command *mode* which can be executed by 'pressing' button *mode* and selection of the desired mode from a pullup menu. Objects are normally displayed in a standard font on the screen. Colored objects are printed on the screen in a different font. Five different fonts are used to depict the standard font and the four existing colours. One of the colours is used to display the so called *framed* objects (internally stored as strings). The strings can be converted to objects by using the *parse*-command. This parser tries to recognize a certain object represented in a certain mode. Syntactically correct programs are generally represented in the standard font. Defined evaluator-fields must take care of proper colouring of objects.

6.1 Interaction Alternatives

The HHDL specification for the generator PHIL is composed of three layers. The first two layers, the *object* and the *context*-layer describe the objects that can be present in the helix-world and their relation. In the third layer a user-friendly interface can be defined. The standard interface PHS can be used to interact with the environment. Some application-dependent commands and representations must be defined to support the application-user. The problem is that we don't know what kind of user uses the environment to describe hhdl-models. Is the user an experienced helix-user or is he just a beginner? Does the user have any knowledge of Pascal or HHDL? The answer to these questions determine how elaborate or advanced the user-interface must be. Two kinds of

users are considered, a beginner and an experienced hhdl-user.

6.2 An Experienced User

Experienced HHDL users tend to use the tools they are used to work with. They will only use new tools if they have real benefit from it. An environment has some advantages over use of a conventional editor. A great advantage is that model-descriptions created in the environment are always syntactically correct. The environment will immediately inform the user if some syntactic rule is violated. The HHDL compiler will accept the created model-description directly. Furthermore on-line documentation can be provided if desired. All knowledge stored by the meta-programmer will be available to the user. Advanced transformations of objects can be introduced to allow the experienced user to optimize the model source-code or to transform the model-description to a normalized form. A beginner would not use such features.

6.3 An Inexperienced User

We consider the inexperienced user to be a user who has at least some knowledge of Pascal but doesn't know much about HHDL. What do we want the user to learn and how are we going to do this?

6.3.1 An Environment User Manual

A software tool will only be used if it is easy to handle. Phil environments are generally very user-friendly if using the Phil House Style. An on-line description of all commands of the PHS is available by means of the *describe-command*-command.(see section 3.5.4) An interaction manual is delivered with the helix-environment which explaines how to use the mouse, the commands and the planes in the representation-window. A command can be defined which gives on-line explanation of interaction with the environment.

6.3.2 Learning Helix

In the interaction layer of the helix environment specification, an interaction style must be defined which answers to the needs of the environment user. We decided to use the environment for educational purposes because experienced helix users did not really feel the need to use this design-tool. The environment is an ideal tool for beginners to work in, even with the standard PHS. The

HHDL structure is given by the tool and warnings are given if rules are violated. The question now is: in what manner do we support the starting helix user. Three alternatives are outlined:

An Introduction to Helix.
The inexperienced Helix user can be helped by means of an on-line manual which explains the use of all helix objects.

A Standard Design Method.
The environment generates examples by which a standard hhdl design method is introduced. The examples stored by the meta-programmer can be selected by the user by means of an example-command button. HHDL constructions are fully explained by the models given.

Exercise Generation.
The environment generates exercises with various complexity. The internal environment solution is compared with the user-solution. The system provides comment on user-actions.

The alternatives given above are not realized in the present environment due to lack of time. An object and index manual is created and explained in section 6.5.1. An on-line documentation facility to explain how to interact with the environment is provided in the PHS by means of the *describe-command*-command.

Some interaction facilities of the PHS are modified and will be explained. The devices created for Helix are described in the next sections. For explanation of the PHS, see section 3.5.4.

6.4 Creating a Helix pane

A new pane (*top2*) is defined for the default Phil-plane which incorporates some helix-devices (**top2-devices**). These devices are listed in [7].

- *title.* A message device giving the message *Helix operations.*
- *type.* A button device "Type" to enable invocation of the *get-type* command.
  This command is associated with the left mouse button by means of the button keyword *:left.*
- *helix-message.* A message device to give type information.
  The MESSAGE function can be used to give a message to the user. The device-variable **message-device** must be set to *'helix-message* before using it.

- *manual.* A button to enable invocation of the *:left*-button "Manual" command. The manual command is associated with the left mouse button. The command *load-manual* is connected to the right button for debugging purposes.

- *index.* A text device is used for reading the manual index.

Some useful functions are given to interact with defined devices:

- **set-texsw** (*texsw string*) Strings can be written to a text window by means of this function. *string* is a Lisp string and *texsw* is a sunview textwindow pointer. The function *plane-pane* must be used to obtain this pointer.

- **plane-pane** (*plane device-name*) Where *plane* is the variable **plane** which points to the current plane, and *device-name* is *'text* which points to the default text-window.

- **reset-texsw** (*texsw*) Before writing to a text-device it can be cleared by means of the function reset-texsw.

6.5 Helix Commands

The PHS contains an elaborate set of commands. Only two new commands are defined : the "Manual" command and the "Type" command.

6.5.1 The Manual Command

The general structure of the HHDL-language is known to the system and may be manipulated with by the user. The inexperienced user however doesn't really know what hhdl objects are used for. He would like some documentation on these objects which tells him how to use them and when. All class definitions in Philgram allow a *:doc* definition, the definition of a documentation string that which will be displayed on the screen if an instance of this class is selected. What we would like is to obtain manual information a bit more flexible. That's why we introduce a manual command. This command uses an object and an index as input. Output is the information found on the index or, if the index is an empty string, information available about the selected object. Documentation is taken from two global Lisp variables *helix-manual* and *predefined-packages*, where the first is a Lisp association list [7] containing object-information and the latter is a variable that is used to create the symlayer containing predefined package information (see section 5.7). The

documentation found is displayed in the text-window *'text*, which is the text device of the default plane.

6.5.2 The Get-Type Command

A mistake that is often made in model-descriptions is misuse of types. The get-*Type* command helps the user by displaying the result of sending the message *:type* to the selected object on the screen. Commands are defined in the Lisp package "PHIL", because it simplifies plane-device-usage. The Phil internal function *%send* is used to send messages to objects. (all message sending is converted internally to such constructions).

6.6 Helix Modes

The modes *:short*, *:medium* and *:default* are introduced here to represent helix programs. The implementation of these formats are given in [7]. In [7] a default mode property is given to <hhdl_programs>, <module> and <package>.

6.6.1 :short

This mode can be used to represent the top object and only shows a list of defined package and module names. No details are given in this representation. The names defined in this mode will be projected on the module and package names in the modes :medium and :default. Module and Package names can only be changed in this mode.

6.6.2 :medium

Selection of this mode gives a medium representation of modules and packages. Names of functions, procedures, comptypes and subprocesses are represented on the screen to give a general view of a program.

6.6.3 :default

This is the default object editor mode. In this mode all the necessary information of modules and packages is available on the screen and can be edited.

Chapter 7

Conclusions and Suggestions

In the previous chapters, we have specified the language HHDL in the PHIL specification language Philgram, from which an environment is generated. The HHDL-specification contains three major layers:

- ☐ The Object Layer
- ☐ The Context Layer
- ☐ The Interaction Layer

The BNF syntax specification of HHDL, given by [5] can easily be translated into a Philgram Object Layer specification. The specification, given by SILVAR & LISCO, needs some adaptation, because left recursive structures can not be specified in Philgram directly. Validation of the Philgram specification with respect to the original BNF specification can be done by means of the function *generate-syntax*. This function generates a BNF-like specification from a Philgram specification. Most formal programming languages are specified by means of a BNF specification. A BNF to Philgram translator would be a very practical tool for PHIL environment specifiers.

In the Helix Context Layer the relations between object are specified. Type relations and context sensitive constraints are checked by defining evaluators, which maintain the invariant relations defined.

In the Interaction Layer of the HHDL specification the user-interface is defined. A standard style, the Phil House Style is delivered with the generator PHIL. The PHS incorporates many commands which can be used to interact with the objects defined in the Helix-world.

On the whole it can be stated that it is easy to specify a language like Pascal or HHDL in the Phil generator specification language Philgram. The generator principle makes it very easy to modify structure and interaction in the environment. The standard user interface is very convenient to work with.

By defining the object layer, an object oriented editor is created for the environment user. The user may manipulate objects instead of text. It is however possible to edit the textual representation of objects. In the context layer many context checks are defined. The result of this is that the environment user gets messages and warnings, and objects are coloured, i.e. they are represented in a special font, if some rule is violated. Model descriptions created in the Helix environment are always syntactically correct, which means that the HHDL compiler will immediately accept the description. The user doesn't have to edit the description over and over, to remove all syntax errors. The interaction layer defined, is a multiple window system. The object tree (the HHDL model descriptions) is represented in a SUNVIEW window. An object can be represented in more than one window, and in different representation modes. In Phil environments there is a notion of a current (selected) object. The user may perform some action on an object by means of a command. A command is executed by selecting a button device, using the mouse, or by selecting it from a menu. The commands defined in the Phil House Style can be used to interact with the environment.

The Helix Environment is very well suited for use by inexperienced as well as by experienced Helix users. For both users an on-line environment user manual is given by means of the *describe-command* command. For inexperienced HHDL users, on-line HHDL object documentation is available.

The Helix environment is an ideal tool for HHDL users to work in. The system gives on-line information on objects and gives warnings and messages when syntactic rules are violated. We would like to do more for the users of the environment. We would like the environment to give an on-line HHDL or even a Helix course. Three alternatives are given by section 6.3.2. The system created now is well suited to build an educational layer on. More research is needed to find the appropriate layer information.

Appendix A

Philgram

In addition to the terminals appearing in a **bold** font, Common Lisp types are used as terminals in the definition of PHILGRAM, the Phil Grammar. All Common Lisp types appear in CAPITALS. Their exact definition can be found in the Common Lisp manual. The following Common Lisp types are used:

- SYMBOL, a the Common Lisp *symbol*,
- KEYWORD, a Common Lisp *symbol* in the *keyword package*,
- FORM, a arbitray Common Lisp form, i.e the Common Lisp type *T*,
- STRING, a Common Lisp *string*,
- CL-ARGLIST, a Common Lisp *argument list*,
- TYPE-SPECIFIER, a Common Lisp *type specifier*,
- FUNCTION-SPECIFIER, a Common Lisp *function specifier*,
- CHARACTER, a Common Lisp type *character*,
- the type specifiers (INTEGER 0 *), specifying a positive integer and the type specifier (VECTOR CHARACTER) specifying a *vector* with elements of type *character*.

Defining constants.

def-constant ::= (**def-constant** *class-name-spec*
[**:doc** *documentation*]
[**:format** *constant-format-spec*]
[**:methods** {*method-spec*}*])
class-name-spec ::= *class-name* | (*class-name* {*prop-name prop-value*}*)
class-name ::= SYMBOL
prop-name ::= KEYWORD
prop-value ::= FORM
documentation ::= STRING

constant-format-spec ::= *simple-string* | *case-string*
simple-string ::= STRING
case-string ::= (**:case** *simple-string*)
method-spec ::= (*message arglist body*)
message ::= KEYWORD
arglist ::= CL-ARGLIST
body ::= {FORM}*

Defining primitives.

def-primitive ::= (**def- primitive** *class-name-spec primitive-spec*
[**:doc** *documentation*]
[**:format** *primitive-format-spec*]
[**:methods** {*method-spec*}*])
primitive-spec ::= (*type-spec* [**:construct** *function-spec*]
[**:copy** *function-spec*]
[**:destruct** *function-spec*])
type-spec ::= TYPE-SPECIFIER
function-spec ::= FUNCTION-SPECIFIER
primitive-format-spec ::= ([**:print** *function-spec*]
[**:parse** *micro-syntax*]
[**:create** *function-spec*])
micro-syntax ::= ({*micro-clause*}*)
micro-clause ::= *character* |
char-predicate |
simple-string |
case-string |
micro-optional |
micro-sequence |
micro-alternative |
micro-syntax
character ::= CHARACTER
char-predicate ::= FUNCTION-SPECIFIER
micro-optional ::= (**:opt** {*micro-clause*}*)
micro-sequence ::= (**:seq** *micro-clause* [**:minimum** *number*])
number ::= (INTEGER 0 *)
micro-alternative ::= (**:alt** {*micro-clause*}*)

Defining structures

def-structure ::= (**def-structure** *class-name-spec structure-spec*
[**:doc** *documentation*]
[**:format** *structure-format-spec*]
[**:methods** {*method-spec*}*])
structure-spec ::= ({*field*}*)
field ::= (*label name* [**:equal** *evaluator-spec*])
label ::= SYMBOL

name ::= *sort-name* | *class-name*
evaluator-spec ::= FORM
structure-format-spec ::= ({*structure-clause*}*)
structure-clause ::= *character* |
simple-string |
case-string |
object-ref |
spaces |
newlines |
structure-indent |
structure-optional |
structure-field |
structure-format-spec
object-ref ::= SYMBOL
spaces ::= (**:sp** *number*)
newlines ::= (**:sp** *number*)
structure-indent ::= (**:indent** {*structure-clause*}+)
structure-optional ::= (**:optional** *object-ref* {*structure-clause*}+)
structure-field ::= (**:field** *length position object-ref*)
length ::= (INTEGER 0 *)
position ::= **:left** | **:right** | **:center**
def-field ::= (**def-field** (*class-name label*) *sort-name*)
[**:equal** *evaluator-spec*])

Defining sequences.

def-sequence ::= (**def-sequence** *class-name-spec sequence-spec*
[**:doc** *documentation*]
[**:format** *sequence-format-spec*]
[**:methods** {*method-spec*}*])
sequence-spec ::= (*name* [**:minimum** *number*])
sequence-format-spec ::= ([**:prologue** *seq-prologue*]
[**:indent** *seq-indent*]
[**:element** *seq-element*]
[**:separator** *seq-separator*]
[**:epilogue** *seq-epilogue*])
seq-prologue ::= ({*seq-layout*}*)
seq-indent ::= **T** | **NIL**
seq-element ::= (*number position*) | **NIL**
seq-separator ::= ({*seq-layout*}*)
seq-epilogue ::= ({*seq-layout*}*)
seq-layout ::= *simple-string* | *case-string* | *spaces* | *newlines* |

Defining sorts.

def-sort ::= (**def-sort** *sort-name-spec sort-spec* [**:doc** *documentation*]
sort-name-spec ::= *sort-name* | (*sort-name* {*prop-name prop-value*}*)
sort-name ::= SYMBOL
sort-spec ::= ({*name*}*)

Defining properties, methods, variables functions.

def-prop ::= (**def-prop** *name prop-name prop-value*)
def-method ::= (**def-method** (*class-name message*) *arglist body*)
def-variable ::= (**def-variable** *variable-name* FORM)
variable-name ::= SYMBOL
def-function ::= (**def-function** *function-name arglist body*)
function-name ::= SYMBOL

Defining formats and parse functions and operators.

def-format ::= (**def-format** (*class-name mode-spec*) *format-spec*)
mode-spec ::= KEYWORD
format-spec ::= *constant-format-spec* |
primitive-format-spec |
structure-format-spec |
sequence-format-spec
def-parser ::= (**def-parser** (*class-name mode-spec*)
optimize-form)
optimize-form ::= *alternatives* |
right-recursive |
left-recursive
alternatives ::= (**alternatives** { class-name }*)
right-recursive ::= (**right-recursive** *label-sort-pair*
oper-form label-sort-pair)
left-recursive ::= (**left-recursive** *label-sort-pair*
oper-form [*label-sort-pair*])
label-sort-pair ::= (*label name*)
oper-form ::= *label-sort-pair* |
simple-string
def-operator ::= (**def-operator** (*class-name sort-name*)
op-type op-priority [*op-assoc*])
op-type ::= **:infix** | **:prefix** | **:postfix**
op-priority ::= *integer*
op-assoc ::= **:left** | **:left**

Defining commands, styles and checks.

def-command ::= (**def-command** *command-name arglist body*)
command-name ::= SYMBOL
def-style ::= (**def-style** *style-name*)
style-name ::= SYMBOL
in-style ::= (**in-style** *style-name &rest comkeys*)
comkeys ::= {*command-name key-sequence*}*
name-string ::= STRING

key-sequence ::= (VECTOR CHARACTER)
def-check ::= (**def-check** *check-name &rest labels*)
check-name ::= SYMBOL
labels ::= {KEYWORD}*

Appendix B

Helix Syntax Specification

This syntax specification of HHDL is generated from the Philgram specification of HHDL by means of the function *generate-syntax*, and looks like a kind of BNF specification.

```
Syntax of <HHDL_PROGRAMS>.

Mode = :DEFAULT


<ABS> ::= "Abs(" <EXPRESSION> ")"  .

<ADDING_SIMPLE> ::= <SIGNED_TERM> <ADD_OPERATOR> <UNSIGNED_SIMPLE_EXP>  .

<ADD_OPERATOR> ::= <PLUS> | <MINUS> | <OR_OPERATOR>  .

<ADD_SIMPLE> ::= <TERM> <ADD_OPERATOR> <UNSIGNED_SIMPLE_EXP>  .

<AND-OPERATOR> ::= "AND" .

<ARCTAN> ::= "Arctan(" <EXPRESSION> ")"  .

<ARRAY_INDICATOR> ::= "[" <INDEXES_LIST> "]"  .

<ARRAY_TYPE> ::= "ARRAY" "[" <INDEX_LIST> "]" "OF" <TYPE>  .

<ASSIGN_STAT> ::= <VARIABLE_LIST> ":=" <EXPRESSION>  .

<BITPACK> ::= "BitPack" .

<BLOCK> ::= {<BLOCK_PART> }1 .

<BLOCK_PART> ::= [ <LABEL_DECL_PART> ]
```

```
                    [ <CONST_DEF_PART> ]
                    [ <TYPE_DEF_PART> ]
                    [ <VAR_DECL_PART> ]
                    [ <PROC_FUNC_DECL_PART> ]
                    <COMPOUND_STAT>  .

<BOOLEAN_TYPE> ::= "BOOLEAN" .

<BOTH_NET_REF_LIST> ::= "BOTHWAYS" {<NETTYPE_DEF> ";" }1 ";" .

<CASE_ELEMENT> ::= <CASE_LABEL_LIST> ":" <STATEMENT>  .

<CASE_ELEMENT_LIST> ::= {<CASE_ELEMENT> ";" }1 .

<CASE_LABEL_LIST> ::= {<CONSTANT> "," }1 .

<CASE_OF_STAT> ::= "CASE" <EXPRESSION>
                      "OF" [ <CASE_ELEMENT_LIST> ]
                           [ ";" <OTHERWISE_CLAUSE> ]
                     "END"  .

<CBLOCK> ::= {<CBLOCK_PART> }1 .

<CBLOCK_PART> ::= [ <DEFAULT_PART> ]
                  [ <NET_REF_PART> ]
                  [ <LABEL_DECL_PART> ]
                  [ <CONST_DEF_PART> ]
                  [ <TYPE_DEF_PART> ]
                  [ <VAR_DECL_PART> ]
                  [ <PROC_FUNC_DECL_PART> ]
                  [ <SUBP_DECL_PART> ]
                  <MODEL_COMPOUND_STAT> ";"  .

<CHAR> ::= (#\' (:ALT ALPHANUMERICP #\~ #\' #\! #\@ #\# #\$ #\% #\^ ...)
#\') .

<CHAR_TYPE> ::= "CHAR" .

<CHECK_LIST> ::= "CHECK" <CHECK_NET_LIST>  .

<CHECK_NET_LIST> ::= {<IDENT> "," }1 .

<CHR> ::= "Chr(" <EXPRESSION> ")"  .

<CLOCK_DEF> ::= "DEFAULT" .

<CLOCK_DEF_PART> ::= "CLOCK" {<CLOCK_LIST> ";" }1 ";" .
```

```
<CLOCK_LIST> ::= <IDENT> "(" <INTEGER> "," <INTEGER> ")" [ <CLOCK_DEF> ]
.

<COMMENT> ::= (#\( #\* (:ALT #\* #\) (:SEQ STANDARD-CHAR-P))) .

<COMPOUND_STAT> ::= "BEGIN" <STATEMENT_LIST> "END" .

<COMPTYPE> ::= "COMPTYPE" <IDENT> [ <FORMAL_PARMS_LIST> ] ";" <CBLOCK> .

<COMP_DEF_PART> ::= <COMPTYPE> | <TRANSLATOR> .

<CONCATENATION_OPERATOR> ::= "//" .

<COND_STAT> ::= <IF_THEN_ELSE_STAT> | <CASE_OF_STAT> .

<CONSTANT> ::= <SIGNED_NUM> | <UNSIGNED_NUM> | <SIGNED_IDENT> | <IDENT> |
               <CHAR> | <STRING> | <TRUE> | <FALSE> .

<CONSTANT_LIST> ::= "," {<CONSTANT> "," }1 .

<CONST_DEF> ::= <IDENT> "=" <CONSTANT> .

<CONST_DEF_PART> ::= "CONST" {<CONST_DEF> ";" }1 ";" .

<COS> ::= "Cos(" <EXPRESSION> ")" .

<CURRTIME> ::= "CurrTime(" <IDENT> ")" .

<DEFAULT_DECL> ::= <IDENT> "=" <CONSTANT> .

<DEFAULT_PART> ::= "DEFAULT" {<DEFAULT_DECL> ";" }1 ";" .

<DELAY> ::= "DELAY" <EXPRESSION> .

<DESENSITIZE_STAT> ::= "DESENSITIZE(" <CHECK_NET_LIST> ")" .

<DETACH_STAT> ::= "DETACH" .

<DIV-OPERATOR> ::= "DIV" .

<DIVIDED> ::= "/" .

<DR_PACK> ::= "Dr_Pack" .

<ELEMENT> ::= <SUBRANGE_EXPRESSION> | <EXPRESSION> .

<ELEMENT_LIST> ::= {<ELEMENT> "," }1 .

<EMPTY> ::= "(* EMPTY *)" .
```

```
<ENUMERATED_TYPE> ::= "(" <IDENT_LIST> ")"  .

<EOF> ::= "Eof" [ "(" <VARIABLE> ")" ]  .

<EOLN> ::= "Eoln" [ "(" <VARIABLE> ")" ]  .

<EQUAL> ::= "=" .

<EXP> ::= "Exp(" <EXPRESSION> ")"  .

<EXPRESSION> ::= <REL_EXP> | <SIMPLE_EXP>  .

<EXPRESSION_IO_LENGTH> ::= <EXPRESSION> [ <IO_LENGTH> ]  .

<EXPRESSION_LIST> ::= {<EXPRESSION_IO_LENGTH> "," }1 .

<EXTERN_FUNCTION> ::= "FUNCTION" <IDENT> [ <FORMAL_PARMS> ] ":" <IDENT> ";"
                      "EXTERN" ";"  .

<EXTERN_PROCEDURE> ::= "PROCEDURE" <IDENT> [ <FORMAL_PARMS> ] ";"
                       "EXTERN" ";"  .

<FACTOR> ::= <TRUE> | <FALSE> | <PARENTHESIZED_EXPRESSION> | <SET_DENOTATION>
|
             <NOT_FACTOR> | <CHAR> | <STRING> | <NIL_DENOTATION> |
             <UNSIGNED_NUM> | <STANDARD_FUNCTION> | <FUNCTION_APPL_WITH_PARS>
|
             <VARIABLE>  .

<FALSE> ::= "FALSE" .

<FIELD_LIST> ::= <FIXED_VARIANT_PART> | <FIXED_PART> | <VARIANT_PART>  .

<FIELD_SELECTOR> ::= "." <IDENT>  .

<FIXED_PART> ::= {<RECORD_SECTION> ";" }1 .

<FIXED_VARIANT_PART> ::= <FIXED_PART> ";" <VARIANT_PART>  .

<FORMAL_PARMS> ::= "(" {<FORMAL_PARM_SEC> ";" }1 ")" .

<FORMAL_PARMS_LIST> ::= "(" {<IDENT_LIST__IDENT> ";" }1 ")" .

<FORMAL_PARM_SEC> ::= <IDENT_LIST__IDENT> | <VAR_IDENT_LIST__IDENT>  .

<FORTRAN_FUNCTION> ::= "FUNCTION" <IDENT> [ <FORMAL_PARMS> ] ":" <IDENT>
";"
                       "FORTRAN" ";"  .
```

```
<FORTRAN_PROCEDURE> ::= "PROCEDURE" <IDENT> [ <FORMAL_PARMS> ] ";" "FOR-
TRAN" ";"  .

<FORWARD_FUNCTION> ::= "FUNCTION" <IDENT> [ <FORMAL_PARMS> ] ":" <IDENT>
";"
                       "FORWARD" ";"  .

<FORWARD_PROCEDURE> ::= "PROCEDURE" <IDENT> [ <FORMAL_PARMS> ] ";"
                        "FORWARD" ";"  .

<FOR_DOWNTO_DO_STAT> ::= "FOR" <IDENT> ":=" <EXPRESSION> "DOWNTO" <EXPRESSION>
                         "DO" <STATEMENT>  .

<FOR_TO_DO_STAT> ::= "FOR" <IDENT> ":=" <EXPRESSION> "TO" <EXPRESSION>
                     "DO" <STATEMENT>  .

<FUNCTION> ::= "FUNCTION" <IDENT> [ <FORMAL_PARMS> ] [ ":" <IDENT> ] ";"
<BLOCK> ";"  .

<FUNCTION_APPL_WITH_PARS> ::= <IDENT> "(" <EXPRESSION_LIST> ")"  .

<GOTO_STAT> ::= "GOTO" <INTEGER>  .

<GREATER-OR-EQUAL> ::= ">=" .

<GREATER> ::= ">" .

<HHDL_PROGRAMS> ::= "=============================="
                    "         HHDL-PROGRAMS        "
                    "=============================="
                    [ <HHDL_PROGS> ]  .

<HHDL_PROGS> ::= {<PROG> }1 .

<IDENT> ::= (ALPHA-CHAR-P (:SEQ (:ALT ALPHANUMERICP #\_))) .

<IDENT_LIST> ::= {<IDENT> "," }1 .

<IDENT_LIST__IDENT> ::= <IDENT_LIST> ":" <IDENT>  .

<IF_THEN_ELSE_STAT> ::= "IF" <EXPRESSION>
                        "THEN" <STATEMENT>
                      [ "ELSE" <STATEMENT> ]  .

<IN-OPERATOR> ::= "IN" .

<INDEXES_LIST> ::= {<EXPRESSION> "," }1 .
```

```
<INDEX_LIST> ::= {<SIMPLE_TYPE> "," }1 .

<INTEGER> ::= (:SEQ DIGIT-CHAR-P :MINIMUM 1) .

<INTEGER_TYPE> ::= "INTEGER" .

<INT_NET_REF_LIST> ::= "INTERNAL" {<NETTYPE_DEF> ";" }1 ";" .

<IN_NET_REF_LIST> ::= "INWARD" {<NETTYPE_DEF> ";" }1 ";" .

<IOPACK> ::= "IoPack" .

<IO_LENGTH> ::= ":" <INTEGER>  .

<LABELLED_STAT> ::= <INTEGER> ":" <UNLABELLED_STAT>  .

<LABEL_DECL_PART> ::= "LABEL" {<INTEGER> "," }1 ";" .

<LASTTIME> ::= "LastTime(" <IDENT> ")"  .

<LESS-OR-EQUAL> ::= "<=" .

<LESS> ::= "<" .

<LN> ::= "Ln(" <EXPRESSION> ")"  .

<LOGPACK> ::= "LogPack" .

<MBLOCK> ::= {<MBLOCK_PART> ";" }1 .

<MBLOCK_PART> ::= [ <USE_PART> ]
                  [ <LABEL_DECL_PART> ]
                  [ <CONST_DEF_PART> ]
                  [ <TYPE_DEF_PART> ]
                  [ <NETTYPE_DECL_PART> ]
                  [ <VAR_DECL_PART> ]
                  [ <PROC_FUNC_DECL_PART> ]
                  <COMPOUND_STAT>  .

<MINUS-SIGN> ::= "-" .

<MINUS> ::= "-" .

<MOD-OPERATOR> ::= "MOD" .

<MODBLOCK> ::= {<MODBLOCK_PART> }1 .

<MODBLOCK_PART> ::= [ <USE_PART> ]
                    [ <TIME_UNIT_STAT> ]
```

```
                    [ <CONST_DEF_PART> ]
                    [ <TYPE_DEF_PART> ]
                    [ <NETTYPE_DECL_PART> ]
                    [ <CLOCK_DEF_PART> ]
                    <COMP_DEF_PART>  .

<MODEL_COMPOUND_STAT> ::= "BEGIN" <MODEL_STATEMENT_LIST> "END"  .

<MODEL_LABELLED_STAT> ::= <INTEGER> ":" <MODEL_UNLABELLED_STAT>  .

<MODEL_STATEMENT> ::= <MODEL_LABELLED_STAT> | <MODEL_UNLABELLED_STAT>  .

<MODEL_STATEMENT_LIST> ::= {<MODEL_STATEMENT> ";" }1 .

<MODEL_STRUCTURED_STAT> ::= <MODEL_COMPOUND_STAT> | <COND_STAT> |
                            <REPETITIVE_STAT> | <WITH_STAT>  .

<MODEL_UNLABELLED_STAT> ::= <MODEL_STRUCTURED_STAT> | <ASSIGN_STAT> |
                            <GOTO_STAT> | <PREDEFINED_PROCEDURE> | <PROC_STAT>
|
                            <NETASSIGN_STAT> | <WAITFOR_STAT> |
                            <SENSITIZE_STAT> | <DESENSITIZE_STAT> |
                            <DETACH_STAT> | <COMMENT> | <EMPTY>  .

<MODULE> ::= "MODULE" <IDENT> ";" <MODBLOCK>  .

<MULT_FACTOR> ::= <FACTOR> <MULT_OPERATOR> <TERM>  .

<MULT_OPERATOR> ::= <CONCATENATION_OPERATOR> | <TIMES> | <DIVIDED> |
                    <DIV-OPERATOR> | <MOD-OPERATOR> | <AND-OPERATOR>  .

<NETASSIGN_STAT> ::= "ASSIGN" [ <TRANSPORT> ] <INDEXES_LIST> "TO" <NET_ID_LIST>
                             [ <TIMING_CLAUSE> ]  .

<NETTYPE_DECL> ::= <IDENT> "=" <TYPE>  .

<NETTYPE_DECL_PART> ::= "NETTYPE" {<NETTYPE_DECL> ";" }1 ";" .

<NETTYPE_DEF> ::= <IDENT_LIST> ":" <IDENT>  .

<NET_ID_LIST> ::= {<IDENT> "," }1 .

<NET_REF_PART> ::= [ <IN_NET_REF_LIST> ]
                   [ <OUT_NET_REF_LIST> ]
                   [ <BOTH_NET_REF_LIST> ]
                   [ <INT_NET_REF_LIST> ]  .

<NEW> ::= "New(" <VARIABLE> [ <CONSTANT_LIST> ] ")"  .
```

```
<NIL_DENOTATION> ::= "NIL" .

<NOT-EQUAL> ::= "<>" .

<NOT_DENOTATION> ::= "NOT" .

<NOT_FACTOR> ::= <NOT_DENOTATION> <FACTOR>  .

<ODD> ::= "Odd(" <EXPRESSION> ")"  .

<ORD> ::= "Ord(" <EXPRESSION> ")"  .

<OR_OPERATOR> ::= "OR" .

<OTHERWISE_CLAUSE> ::= "OTHERWISE" <STATEMENT> [ <SEMICOLON> ]  .

<OUT_NET_REF_LIST> ::= "OUTWARD" {<NETTYPE_DEF> ";" }1 ";" .

<PACK> ::= <IDENT> | <REG_PACK> | <BITPACK> | <LOGPACK> | <UTL_PACK> |
                     <DR_PACK> | <RND_PACK> | <STRPACK> | <IOPACK>  .

<PACKAGE> ::= "PACKAGE" <IDENT> ";" <MBLOCK> "."  .

<PAGE> ::= "Page(" <VARIABLE> ")"  .

<PARENTHESIZED_EXPRESSION> ::= "(" <EXPRESSION> ")"  .

<PLUS-SIGN> ::= "+" .

<PLUS> ::= "+" .

<POINTER_DEREFERENCE> ::= "^" .

<POINTER_TYPE> ::= "^" <IDENT>  .

<PRED> ::= "Pred(" <EXPRESSION> ")"  .

<PREDEFINED_FUNCTION> ::= <CURRTIME> | <RECALL> | <LASTTIME> | <ABS> | <SQR>
|
                          <SQRT> | <SIN> | <COS> | <ARCTAN> | <EXP> | <LN>
|
                          <ODD> | <EOLN> | <EOF> | <TRUNC> | <ROUND> | <ORD>
|
                          <SUCC> | <PRED> | <CHR>  .

<PREDEFINED_PROCEDURE> ::= <NEW> | <RESET> | <REWRITE> | <PAGE> | <READ>
|
                           <WRITE> | <READLN> | <WRITELN>  .
```

```
<PROCEDURE> ::= "PROCEDURE" <IDENT> [ <FORMAL_PARMS> ] ";" <BLOCK> ";"  .

<PROC_FUNC> ::= <PROCEDURE> | <FUNCTION> | <FORWARD_PROCEDURE> |
                <FORWARD_FUNCTION> | <EXTERN_PROCEDURE> | <EXTERN_FUNCTION>
|
                <FORTRAN_PROCEDURE> | <FORTRAN_FUNCTION>  .

<PROC_FUNC_DECL_PART> ::= {<PROC_FUNC> }1 .

<PROC_STAT> ::= <IDENT> [ "(" <EXPRESSION_LIST> ")" ]  .

<PROG> ::= <PROGRAM>  .

<PROGRAM> ::= <MODULE> | <PACKAGE>  .

<QUALIFIER> ::= <CHECK_LIST> | <TIMING_CLAUSE>  .

<READ> ::= "Read(" <VARIABLE_LIST> ")"  .

<READLN> ::= "Readln" [ "(" <VARIABLE_LIST> ")" ]  .

<REAL> ::= ((:SEQ DIGIT-CHAR-P :MINIMUM 1) (:ALT (#\. # #) ("e" # #))) .

<REAL_TYPE> ::= "REAL" .

<RECALL> ::= "Recall(" <IDENT> ")"  .

<RECORD_SECTION> ::= <IDENT_LIST> ":" <TYPE>  .

<RECORD_TYPE> ::= "RECORD" [ <FIELD_LIST> ] "END"  .

<REC_VAR_LIST> ::= {<VARIABLE> "," }1 .

<REG_IDENT_TYPE> ::= "REGISTER" "[" <IDENT> "]"  .

<REG_PACK> ::= "Reg_Pack" .

<REG_SUBRANGE_TYPE> ::= "REGISTER" "[" <SUBRANGE_TYPE> "]"  .

<REG_TYPE> ::= <REG_SUBRANGE_TYPE> | <REG_IDENT_TYPE>  .

<REL_EXP> ::= <SIMPLE_EXP> <REL_OPERATOR> <SIMPLE_EXP>  .

<REL_OPERATOR> ::= <EQUAL> | <NOT-EQUAL> | <LESS-OR-EQUAL> | <LESS> |
                   <GREATER-OR-EQUAL> | <GREATER> | <IN-OPERATOR>  .

<REPEAT_UNTIL_STAT> ::= "REPEAT" <STATEMENT_LIST> "UNTIL" <EXPRESSION>  .
```

```
<REPETITIVE_STAT> ::= <WHILE_DO_STAT> | <REPEAT_UNTIL_STAT> | <FOR_TO_DO_STAT>
|
                      <FOR_DOWNTO_DO_STAT>  .

<RESET> ::= "Reset(" <VARIABLE> ")"  .

<REWRITE> ::= "Rewrite(" <VARIABLE> ")"  .

<RND_PACK> ::= "Rnd_Pack" .

<ROUND> ::= "Round(" <EXPRESSION> ")"  .

<S-EXP> ::= "Given" .

<SELECTORS> ::= {<VARIABLE_SELECTOR> }1 .

<SEMAPHORE_TYPE> ::= "SEMAPHORE" .

<SEMICOLON> ::= ";" .

<SENSITIZE_STAT> ::= "SENSITIZE(" <CHECK_NET_LIST> ")"  .

<SET_DENOTATION> ::= "[" [ <ELEMENT_LIST> ] "]"  .

<SET_TYPE> ::= "SET" "OF" <SIMPLE_TYPE>  .

<SIGN> ::= <PLUS-SIGN> | <MINUS-SIGN>  .

<SIGNAL_STAT> ::= "SIGNAL(" <IDENT> ")"  .

<SIGNED_IDENT> ::= <SIGN> <IDENT>  .

<SIGNED_NUM> ::= <SIGN> <UNSIGNED_NUM>  .

<SIGNED_TERM> ::= [ <SIGN> ] <TERM>  .

<SIMPLE_EXP> ::= <ADDING_SIMPLE> | <SIGNED_TERM>  .

<SIMPLE_TYPE> ::= <BOOLEAN_TYPE> | <CHAR_TYPE> | <REAL_TYPE> | <INTEGER_TYPE>
|
                  <SEMAPHORE_TYPE> | <TEXT_TYPE> | <ENUMERATED_TYPE> |
                  <SUBRANGE_TYPE> | <IDENT>  .

<SIN> ::= "Sin(" <EXPRESSION> ")"  .

<SQR> ::= "Sqr(" <EXPRESSION> ")"  .

<SQRT> ::= "Sqrt(" <EXPRESSION> ")"  .
```

```
<STANDARD_FUNCTION> ::= <PREDEFINED_FUNCTION>  .

<STATEMENT> ::= <LABELLED_STAT> | <UNLABELLED_STAT>  .

<STATEMENT_LIST> ::= {<STATEMENT> ";" }1 .

<STRING> ::= (#\' (:SEQ (:ALT ALPHANUMERICP #\~ #\` #\! #\@ #\# #\$ #\% #\^
...) :MINIMUM 0) #\') .

<STRPACK> ::= "StrPack" .

<STRUCTURED_STAT> ::= <COMPOUND_STAT> | <COND_STAT> | <REPETITIVE_STAT> |
                      <WITH_STAT>  .

<STRUCTURED_TYPE> ::= <ARRAY_TYPE> | <RECORD_TYPE> | <SET_TYPE> | <REG_TYPE>
.

<SUBP_COMPOUND_STAT> ::= "BEGIN" <SUBP_STATEMENT_LIST> "END"  .

<SUBP_DECL> ::= <UPON_DO> | <TRANSMIT_TO>  .

<SUBP_DECL_PART> ::= "SUBPROCESS" {<SUBP_DECL> }1 .

<SUBP_LABELLED_STAT> ::= <INTEGER> ":" <SUBP_UNLABELLED_STAT>  .

<SUBP_STATEMENT> ::= <SUBP_LABELLED_STAT> | <SUBP_UNLABELLED_STAT>  .

<SUBP_STATEMENT_LIST> ::= {<SUBP_STATEMENT> ";" }1 .

<SUBP_STRUCTURED_STAT> ::= <SUBP_COMPOUND_STAT> | <COND_STAT> |
                           <REPETITIVE_STAT> | <WITH_STAT>  .

<SUBP_UNLABELLED_STAT> ::= <SUBP_STRUCTURED_STAT> | <ASSIGN_STAT> |
                           <GOTO_STAT> | <PREDEFINED_PROCEDURE> | <PROC_STAT>
|
                           <NETASSIGN_STAT> | <WAITFOR_STAT> | <WAIT_STAT>
|
                           <SIGNAL_STAT> | <COMMENT> | <EMPTY>  .

<SUBRANGE_EXPRESSION> ::= <EXPRESSION> ".." <EXPRESSION>  .

<SUBRANGE_INDICATOR> ::= "[" <EXPRESSION> ".." <EXPRESSION> "]"  .

<SUBRANGE_TYPE> ::= <CONSTANT> ".." <CONSTANT>  .

<SUCC> ::= "Succ(" <EXPRESSION> ")"  .

<SYMLAYER> ::= "Given" .
```

```
<SYMSET> ::= "Given" .

<SYNC> ::= "SYNC" [ <IDENT> ] [ "PHASE" <EXPRESSION> ]  .

<TAG_FIELD> ::= [ <IDENT> ":" ] <IDENT>  .

<TERM> ::= <MULT_FACTOR> | <FACTOR>  .

<TEXT> ::= {<STRING> }1 .

<TEXT_TYPE> ::= "TEXT" .

<TIMES> ::= "*" .

<TIME_UNIT_STAT> ::= "TIMEUNITS" <INTEGER> ";"  .

<TIMING_CLAUSE> ::= <DELAY> | <SYNC>  .

<TRANSLATOR> ::= "TRANSLATOR" <IDENT> [ <FORMAL_PARMS_LIST> ] ";" <CBLOCK>
.

<TRANSMIT_TO> ::= <IDENT> ":" "TRANSMIT" <EXPRESSION> <QUALIFIER>
                              "      TO" <VARIABLE> [ <TIMING_CLAUSE> ] ";"
.

<TRANSPORT> ::= "TRANSPORT" .

<TRUE> ::= "TRUE" .

<TRUNC> ::= "Trunc(" <EXPRESSION> ")"  .

<TYPE> ::= <SIMPLE_TYPE> | <STRUCTURED_TYPE> | <POINTER_TYPE>  .

<TYPE_DEF> ::= <IDENT> "=" <TYPE>  .

<TYPE_DEF_PART> ::= "TYPE" {<TYPE_DEF> ";" }1 ";" .

<UNLABELLED_STAT> ::= <STRUCTURED_STAT> | <ASSIGN_STAT> | <GOTO_STAT> |
                      <PREDEFINED_PROCEDURE> | <PROC_STAT> | <NETASSIGN_STAT>
|
                      <WAITFOR_STAT> | <COMMENT> | <EMPTY>  .

<UNSIGNED_NUM> ::= <REAL> | <INTEGER>  .

<UNSIGNED_SIMPLE_EXP> ::= <ADD_SIMPLE> | <TERM>  .

<UPON_DO> ::= <IDENT> ":" "UPON" <EXPRESSION> <QUALIFIER>
                          "  DO" <SUBP_STATEMENT> ";"  .
```

```
<USE_PART> ::= "USE" {<PACK> "," }1 ";" .

<UTL_PACK> ::= "Utl_Pack" .

<VARIABLE> ::= <IDENT> [ <SELECTORS> ]  .

<VARIABLE_LIST> ::= {<VARIABLE> "," }1 .

<VARIABLE_SELECTOR> ::= <ARRAY_INDICATOR> | <POINTER_DEREFERENCE> |
                        <SUBRANGE_INDICATOR> | <FIELD_SELECTOR>  .

<VARIANT_LIST> ::= {<VARIANT_SECTION> ";" }1 ";" .

<VARIANT_PART> ::= "CASE" <TAG_FIELD> "  OF" [ <VARIANT_LIST> ]  .

<VARIANT_SECTION> ::= <CASE_LABEL_LIST> ":" "(" <FIELD_LIST> ")"  .

<VAR_DECL_PART> ::= "VAR" {<VAR_LIST> ";" }1 ";" .

<VAR_IDENT_LIST__IDENT> ::= "VAR" <IDENT_LIST> ":" <IDENT>  .

<VAR_LIST> ::= <IDENT_LIST> ":" <TYPE>  .

<WAITFOR_STAT> ::= "WAITFOR" [ <EXPRESSION> ] <QUALIFIER>  .

<WAIT_STAT> ::= "WAIT(" <IDENT> ")"  .

<WHILE_DO_STAT> ::= "WHILE" <EXPRESSION> "DO" <STATEMENT>  .

<WITH_STAT> ::= "WITH" <REC_VAR_LIST> "DO" <STATEMENT>  .

<WRITE> ::= "Write(" <EXPRESSION_LIST> ")"  .

<WRITELN> ::= "Writeln" [ "(" <EXPRESSION_LIST> ")" ]  .

Keywords: "Writeln" "Write(" "WITH" "WHILE" "WAIT(" "WAITFOR" "VAR" "  OF"
 "Utl_Pack" "USE" "  DO" "UPON" "TYPE" "Trunc(" "TRUE" "TRANSPORT" "      TO"
 "TRANSMIT" "TRANSLATOR" "TIMEUNITS" "*" "TEXT" "PHASE" "SYNC" "Succ(" ".."
"SUBPROCESS" "StrPack" "Sqrt(" "Sqr(" "Sin(" "SIGNAL(" "SET" "SENSITIZE("
"SEMAPHORE" "Round(" "Rnd_Pack" "Rewrite(" "Reset(" "UNTIL" "REPEAT" "Reg_Pack"
 "REGISTER" "RECORD" "Recall(" "REAL" "Readln" "Read(" "Pred(" "^" "+" "Page("
 "PACKAGE" "OUTWARD" "OTHERWISE" "OR" "Ord(" "Odd(" "NOT" "<>" "NIL" "New("
 "NETTYPE" "ASSIGN" "MODULE" "MOD" "-" "LogPack" "Ln(" "<" "<=" "LastTime("
"LABEL" "IoPack" "INWARD" "INTERNAL" "INTEGER" "IN" "ELSE" "THEN" "IF"
 "         HHDL-PROGRAMS         " "==============================" ">" ">="
 "GOTO" "TO" "DO" "DOWNTO" "FOR" "FORWARD" "FORTRAN" "." "FALSE" "PROCEDURE"
"EXTERN" "FUNCTION" "Exp(" "Eoln" "Eof" "(* EMPTY *)" "Dr_Pack" "/" "DIV"
"DETACH" "DESENSITIZE(" "DELAY" "CurrTime(" "Cos(" "CONST" "=" "//" "COMPTYPE"
```

```
"BEGIN" "(" "CLOCK" "DEFAULT" "Chr(" "CHECK" "CHAR" "END" "CASE" "," ":"
";"
"BOTHWAYS" "BOOLEAN" "BitPack" ":=" "OF" "ARRAY" "]" "[" "Arctan(" "AND"
")"
 "Abs("
```

Appendix C

The Method :ptype.

Ptype methods are defined for all objects of class <TYPE>. Objects off this class are listed below, followed by the information they must return if they receive the message *:ptype*. The information returned is a complete type-specification list.

<ENUMERATED_TYPE> : ('ENUMERATED "ident1" "ident2" ...)
where "ident1" "ident2" ... are unique enumerated constants of this type.

<BOOLEAN_TYPE> : ('BOOLEAN)
<CHAR_TYPE> : ('CHAR)
<REAL_TYPE> : ('REAL)
<INTEGER_TYPE> : ('INTEGER)
<SEMAPHORE_TYPE> : ('SEMAPHORE)
<TEXT_TYPE> : ('TEXT)

<SUBRANGE_TYPE> : ('SUBRANGE (<INTEGER_TYPE> <INTEGER_TYPE>)) |
('SUBRANGE (<CHAR_TYPE> <CHAR_TYPE>)) |
('SUBRANGE (<ENUMERATED_TYPE>
<ENUMERATED_TYPE>))
where both enumerated types are the same type.

<POINTER_TYPE> : ('POINTER "ident")
where "ident" is a type specifier.

<SET_TYPE> : ('SET <ENUMERATED_TYPE>) |
('SET <SUBRANGE_TYPE>) |
('SET <INTEGER_TYPE>) |
('SET <CHAR_TYPE>)

<RECORD_TYPE> : ('RECORD field-list)

<REGISTER_TYPE> : ('REGISTER type-list)
where type-list specifies a subrange type.

<ARRAY_TYPE> :	('ARRAY (simple-type*) component-type) where simple-type is a subrange type.
<IDENT> :	("identifier") where "identifier" is a type specifier.

Appendix D

Typed Objects.

Some objects in HHDL are of a certain abstract data type. These objects are listed below. Some of these objects are sorts specifying a set of classes. Elements of this set are objects of an abstract data type too.

<unsigned_simple_exp> ::= <add_simple> | (structure)
<term>

<term> ::= <mult_factor> | (structure)
<factor>

<factor> ::= <parenthesized_expression> | (structure)
<set_denotation> | (structure)
<not_factor> | (structure)
<string> | (primitive)
<nil_denotation> | (constant)
<unsigned_num> |
<function_appl_with_pars> | (structure)
<variable> | (structure)
<true> | (constant)
<false> | (constant)
<char> | (primitive)
<standard_function> (structure)

<unsigned_num> ::= <real> | (primitive)
<integer> (primitive)

<expression> ::= <rel_exp> | (structure)

<simple_exp>

<simple_exp> ::=	<adding_simple> |	(structure)
	<signed_term>	(structure)

<constant> ::=	<signed_num> |	(structure)
	<unsigned_num> |	
	<signed_ident> |	(structure)
	<ident> |	(primitive)
	<string> |	(primitive)
	<true> |	(constant)
	<false> |	(constant)
	<char>	(primitive)

Appendix E

The :ctype Methods.

For some objects, that are of a certain abstract data type, a so called *:ctype* method is defined. The method *:ctype* for the objects listed below, returns the object-type if it is a HHDL constant.

<SIGNED_NUM>	: return :ctype of sub-object <UNSIGNED_NUM>
<SIGNED_IDENT>	: return :ctype of sub-object <IDENT>
<SIGNED_TERM>	: return :ctype of sub-object <TERM>
<REAL>	: return ('REAL)
<INTEGER>	: return ('INTEGER)
<STRING>	: return ('STRING)
<CHAR>	: return ('CHAR)
<TRUE>	: return ('BOOLEAN)
<FALSE>	: return ('BOOLEAN)
<IDENT>	: look in symlayers for a constant this constant can be an identifier, if so , look up identifier, must be a constant

Appendix F

The :vtype Methods.

For object of the classes listed below, a *:vtype* method is defined. This method is used to obtain the type of HHDL variable. Only the objects listed below will respond to the *:vtype* message.

<SIGNED_IDENT>	: return :vtype of sub-object <IDENT>
<SIGNED_TERM>	: return :vtype of sub-object <TERM>
<IDENT>	: look in symlayers for a variable
<VARIABLE> ::= <IDENT> <SELECTORS>	: look in symlayers for a variable the selectors can be :
<POINTER_DEREFERENCE>	: <IDENT> must be of pointer type
<FIELD_SELECTOR>	: <IDENT> must be of record type
<ARRAY_INDICATOR>	: <IDENT> must be of array type
<SUBRANGE_INDICATOR>	: <IDENT> must be of register type

Appendix G

The :type Methods.

To obtain the type of all objects of some kind of abstract HHDL data type, the *:type* methods are defined. The objects for which such a *:type* method must be defined, and the information they have got to return, are listed below.

<SIGNED_TERM>	: return :type of sub-object <TERM>
<SIGNED_NUM>	: return :type of sub-object <UNSIGNED_NUM>
<PARENTHESIZED_EXPRESSION>	: return :type of sub-object <EXPRESSION>
<NOT_FACTOR>	: return ('BOOLEAN)
<NIL_DENOTATION>	: return ('NILTYPE)
<REAL>	: return ('REAL)
<INTEGER>	: return ('INTEGER)
<STRING>	: return ('STRING)
<CHAR>	: return ('CHAR)
<TRUE>	: return ('BOOLEAN)
<FALSE>	: return ('BOOLEAN)
<IDENT>	: look in symlayers for - a constant, return :ctype - a variable, return :vtype - a function without parms, return type - a pin, return type-info of type-spec

oper	type(N1)	type(N2)	type(Obj)	**Obj ::= N1 oper N2**
"OR"	BOOLEAN	BOOLEAN	BOOLEAN	
"+"/"-"	REAL	REAL	REAL	
	INTEGER	REAL	REAL	
	REAL	INTEGER	REAL	
	INTEGER	INTEGER	INTEGER	
	SET	SET	SET	same base-types or empty sets

Table G.1: The Add(ing)-simple type.

oper	type(N1)	type(N2)	type(Obj)	**Obj ::= N1 oper N2**
"*"	REAL	REAL	REAL	
	INTEGER	REAL	REAL	
	REAL	INTEGER	REAL	
	INTEGER	INTEGER	INTEGER	
	SET	SET	SET	same base-types or empty sets
"/"	INTEGER	INTEGER	REAL	
	INTEGER	REAL	REAL	
	REAL	INTEGER	REAL	
	REAL	REAL	REAL	
"AND"	BOOLEAN	BOOLEAN	BOOLEAN	
"//"	REGISTER	REGISTER	REGISTER	
"DIV"/"MOD"	INTEGER	INTEGER	INTEGER	

Table G.2: The Mult-factor type.

<VARIABLE> : look in symlayers for
- a constant, return :ctype
- a variable, return :vtype
- a function without parms, return type
- a pin, return type-info of type-spec

An Add(ing)-simple looks like : **Obj ::= N1 oper N2**. The allowed sub-object types and the result type are given by table G.1.

A Mult-factor looks like : **Obj ::= N1 oper N2**. The types returned by the *:type* methods are given by table G.2.

oper	type(N1)	type(N2)	**Obj ::= N1 oper N2**
"<"/">"	INTEGER	INTEGER	
	INTEGER	REAL	
	REAL	INTEGER	
	REAL	REAL	
	STRING	STRING	
	BOOLEAN	BOOLEAN	
"="/"<>"/">="/"<="	INTEGER	INTEGER	
	INTEGER	REAL	
	REAL	INTEGER	
	REAL	REAL	
	STRING	STRING	
	BOOLEAN	BOOLEAN	
	SET	SET	
"IN"	type(N1)	SET	type(N1) is basetype of SET

Table G.3: The Rel-expression type.

The type of a relative expression is always boolean. Allowed sub-object-types are given by table G.3.
The expression looks like : **Obj ::= N1 oper N2**.

Appendix H

Type specification checks

In the Helix Object Layer, the global structure of all subobjects of the sort <TYPE> is given. Not all specification are however correct. The type specifications that are allowed are listed in appendix C. For some type specifiers the type specification must be checked. These objects are listed below.

<VAR_LIST>	: type must be checked
<TYPE_DEF>	: type must be checked
<NETTYPE_DECL>	: type must be checked
<CONST_DEF>	: the constant must have a :ctype
<INTEGER_TYPE>	: type is always correct
<REAL_TYPE>	: type is always correct
<CHAR_TYPE>	: type is always correct
<BOOLEAN_TYPE>	: type is always correct
<SEMAPHORE_TYPE>	: type is always correct
<TEXT_TYPE>	: type is always correct
<ENUMERATED_TYPE>	: type is always correct
<IDENT>	: this identifier must be present in the TYPE declaration
<ARRAY_TYPE>	: the index types must be of a correct subrange type : the component type must exist (in TYPE declaration)
<RECORD_TYPE>	: the field types of the fixed and variant part must be checked : the constants of the case-label-lists of the variant sections of the variant-part must be of type tag-type : there may be no double field names
<SUBRANGE_TYPE>	: subrange type must be correct.

	integer, character or enumerated.
<REG_SUBRANGE_TYPE>	: subrange type must be correct
<REG_IDENT_TYPE>	: ident must be a correct subrange type
<SET_TYPE>	: set type must be correct. integer, character, subrange or enumerated.
<POINTER_TYPE>	: identifier in pointer type must be a type-specifier (in TYPE declaration)

Appendix I

Helix Context Checks

The checks that are defined in the Helix Environment are listed below. The actual Philgram definitions can be found in [7]. Some comment on this realization is given in appendix J.

```
<ADDING_SIMPLE> ::= <SIGNED_TERM> <ADD_OPERATOR> <UNSIGNED_SIMPLE_EXP>
                    <signed_term> and <unsigned_simple_exp> must be of a certain
                    TYPE  for a certain <add_operator>


<ADD_SIMPLE> ::= <TERM> <ADD_OPERATOR> <UNSIGNED_SIMPLE_EXP>
                 <term> and <unsigned_simple_exp> must be of a certain TYPE for
                 a certain <add_operator>


<ASSIGN_STAT> ::= <VARIABLE_LIST> ":=" <EXPRESSION>
                  TYPE of <variable>s of <variable_list> must be
                  TYPE of <expression>


<CASE_OF_STAT> ::= "CASE" <EXPRESSION> "OF" <CASE_ELEMENT_LIST>
                             [ ";" <OTHERWISE_CLAUSE> ] "END"
                   TYPE of <constant>s in <case_label_list>s of <case_element>s
                   of <case_element_list>s must be TYPE of <expression>
                   no double constants in case of statement

<CBLOCK_PART> no double default declarations

<CHECK_NET_LIST> ::= <IDENT> [ {"," <IDENT>} ]
                     <ident>s must be of PIN TYPE

<DEFAULT_DECL> ::= <IDENT> "=" <CONSTANT>
```

```
                TYPE of <constant> must be TYPE of <ident>
                <ident> must have been declared in <formal_parms_list> of
                <comp_def_part>

<DELAY> ::= "DELAY" <EXPRESSION>
        <expression> must be of TYPE integer

<EXTERN_FUNCTION> ::= "FUNCTION" <IDENT> [ <FORMAL_PARMS> ] ":" <IDENT> ";"
              "EXTERN" ";"
                  second <ident> must be an existing TYPE

<FORTRAN_FUNCTION> ::= "FUNCTION" <IDENT> [ <FORMAL_PARMS> ] ":" <IDENT>
";"
                                  "FORTRAN" ";"
                    second <ident> must be an existing TYPE

<FORWARD_FUNCTION> ::= "FUNCTION" <IDENT> [ <FORMAL_PARMS> ] ":" <IDENT>
";"
                                "FORWARD" ";"
                    second <ident> must be an existng TYPE

<FOR_DOWNTO_DO_STAT> ::= "FOR" <IDENT> ":=" <EXPRESSION> "DOWNTO" <EXPRESSION>
                           "DO" <STATEMENT>
          <ident> must be of TYPE                1) integer or real
          <expression>s must be of type integer  2) enumerated
          expressions must be of type enumerated 3) subrange of
                                                    integer or enumerated

<FOR_TO_DO_STAT> ::= "FOR" <IDENT> ":=" <EXPRESSION> "TO" <EXPRESSION>
                "DO" <STATEMENT>
                 <ident> must be of a certain TYPE integer or real
                 TYPE of <expression>s must be INTEGER

<FUNCTION> ::= "FUNCTION" <IDENT> [ <FORMAL_PARMS> ] [ ":" <IDENT> ] ";"
                                  <BLOCK>
           second <ident> must be an existing TYPE if function is not forward
           declared
           check double forward, extern and fortran procs/funcs

<FUNCTION_APPL_WITH_PARS> ::= <IDENT> "(" <EXPRESSION_LIST> ")"
                          function must have been declared
                          LENGTH of <expression_list> must be correct
                          TYPE of <expression>s of <expression_list>
                          must be correct

<GOTO_STAT> ::= "GOTO" <INTEGER>
            <integer> must have been declared as a label
            label must exist
```

```
<HHDL_PROGRAMS> no double program names

<IDENT_LIST__IDENT> ::= <IDENT_LIST> ":" <IDENT>
                        <ident> must be an existing TYPE

<IF_THEN_ELSE_STAT> ::= "IF" <EXPRESSION> "THEN" <STATEMENT>
"ELSE" <STATEMENT> ]
                        <expression> must be of TYPE boolean

<LABELLED_STAT> ::= <INTEGER> ":" <UNLABELLED_STAT>
                    <integer> must have been declared as a label
                    label may only be present once

<MODEL_LABELLED_STAT> ::= <INTEGER> ":" <MODEL_UNLABELLED_STAT>
                          <integer> must have been declared as a label
                          label may only be present once

<MODULE> no double comptype names
         no double subproces names
         check double forward, extern and fortran procs/funcs

<MULT_FACTOR> ::= <FACTOR> <MULT_OPERATOR> <TERM>
                  <factor> and term must be of a certain TYPE for a certain
                  <mult_operator>

<NETASSIGN_STAT> ::= "ASSIGN" [ <TRANSPORT> ] <INDEXES_LIST> "TO" <NET_ID_LIST>
                                [ <TIMING_CLAUSE> ]
                     LENGTH of <indexes_list> must be LENGTH of <net_id_list>
                     TYPE of <expression>s of <indexes_list> must be
                     TYPE of <ident>s of <net_id_list>
                     no double identifiers in netassign-stat

<NETTYPE_DEF> ::= <IDENT_LIST> ":" <IDENT>
                  <ident> must be an existing NETTYPE

<NOT_FACTOR> ::= <NOT_DENOTATION> <FACTOR>
                 <factor> must be of TYPE boolean

<PACKAGE> check double forward, extern and fortran procs/funcs

<PROCEDURE> check double forward, extern and fortran procs/funcs

<PROC_STAT> ::= <IDENT> [ "(" <EXPRESSION_LIST> ")" ]
                procedure must have been declared
                LENGTH of <expression_list> (if used) must be correct
                TYPEs of <expression>s f <expression_io_length>s of
                <expression_list> (if used) must be correct

<REL_EXP> ::= <SIMPLE_EXP> <REL_OPERATOR> <SIMPLE_EXP>
```

```
          <simple_exp>s must be of a certain TYPE for a certain <rel_operator>

<REPEAT_UNTIL_STAT> ::= "REPEAT" <STATEMENT_LIST> "UNTIL" <EXPRESSION>
                        <expression> must be of TYPE boolean

<SIGNAL_STAT> ::= "SIGNAL(" <IDENT> ")"
                  TYPE of <ident> must be semaphore

<SIGNED_IDENT> ::= <SIGN> <IDENT>
                   TYPE of <ident> must be constant TYPE : REAL or INTEGER

<SIGNED_TERM> ::= [ <SIGN> ] <TERM>
                  if <sign> is used, term must be of  TYPE : REAL or INTEGER

 <SUBP_LABELLED_STAT> ::= <INTEGER> ":" <SUBP_UNLABELLED_STAT>
                          <integer> must have been declared as a label
                          label may only be present once

<SYNC> ::= "SYNC" [ <IDENT> ] [ "PHASE" <EXPRESSION> ]
       <ident> (if used) must be a clock
       <expression> (if used) must be of TYPE integer (>=0)

<TIME_UNIT_STAT> ::= "TIMEUNITS" <INTEGER> ";"
                     + 0 <= <integer> <= 15

<TRANSMIT_TO> ::= <IDENT> ":" "TRANSMIT" <EXPRESSION> <QUALIFIER>
      "      TO" <VARIABLE> [ <TIMING_CLAUSE> ] ";"
              TYPE of <expression> must be TYPE of <variable>
      <variable> must be of PIN TYPE

<UPON_DO> ::= <IDENT> ":" "UPON" <EXPRESSION> <QUALIFIER>
                            "  DO" <SUBP_STATEMENT> ";"
          <expression> must be of TYPE boolean

<VAR_IDENT_LIST__IDENT> ::= "VAR" <IDENT_LIST> ":" <IDENT>
                           <ident> must be an existing TYPE

<WAITFOR_STAT> ::= "WAITFOR" [ <EXPRESSION> ] <QUALIFIER>
                   <expression> (if used) must be of TYPE boolean

<WAIT_STAT> ::= "WAIT(" <IDENT> ")"
            <ident> must be of TYPE semaphore

<WHILE_DO_STAT> ::= "WHILE" <EXPRESSION> "DO" <STATEMENT>
                    <expression> must be of TYPE boolean

<WITH_STAT> ::= "WITH" <REC_VAR_LIST> "DO" "BEGIN" <STATEMENT> "END"
            <variable>s of <rec_var_list> must be RECORD variables
            <variable>s in <statement> can be record-fields
```

Allowing just one DEFAULT clock

Checking Standard Functions and Procedures

Argument must be a PIN : <CURRTIME> <RECALL> <LASTTIME>

Argument must be a file variable (TEXT variable) : <EOLN> <EOF>
<RESET> <REWRITE> <PAGE>

Argument must be of type INTEGER or REAL : <ABS> <SQR> <SQRT> <SIN> <COS>
<ARCTAN> <EXP> <LN> <ODD> <TRUNC>
<ROUND>

Argument must be of ordinal type : <ORD> <CHR> <SUCC> <PRED>

<NEW> : Argument must be of pointer type
Tags must be checked

<READ> <READLN> : Types of arguments must be checked

<WRITE> <WRITELN> : Types of arguments must be checked

Appendix J

checks-lisp

This appendix gives some comment on the Philgram realization of the context checks that are defined in the environment, given by appendix I.

<ADDING_SIMPLE> ==>
Allowed sub-object types are given by appendix G.

<ADD_SIMPLE> ==>
Allowed sub-object types are given by appendix G.

<ASSIGN_STAT> ==>
The type variables in the variable-list depend on the type of the expression. See table J.1.

variable	expression	
REAL	REAL	
REAL	INTEGER	
INTEGER	INTEGER	
BOOLEAN	BOOLEAN	
CHAR	CHAR	
INTEGER	INTEGER	subranges of
CHAR	CHAR	subranges of
ENUMERATED	ENUMERATED	subranges of
SET	type(expression)	basetypes are equal or empty sets

Table J.1: Assignment statement.

Assignments of a value to a subrange type variable that exceed the limit of the subrange may or may not result in an error. This depends on the host-computer system. Range-checks are therefore not defined in the context layer specification of HHDL. Wrong assignments should however be detected by the environment. Range-check evaluators should therefore be defined.

<CASE_OF_STAT> ==>

identifier	expressions	
INTEGER or REAL	INTEGER	
ENUMERATED	ENUMERATED	
INTEGER	INTEGER	subranges of
ENUMERATED	ENUMERATED	subranges of

Table J.2: For-(down)to-do statement.

The *:ctype* result of all constants is compared with the *:type* result of the expression. All constants are stored in a symset object to check double occurrences.

<CBLOCK_PART> ==>
All default comptype/translator parameters are stored in a symset object to check double declarations.

<CHECK_NET_LIST> ==>
All identifiers must be stored in a symlayer as PINs.

<DEFAULT_DECL> ==>
The identifier must be a retrievable. Such parameters are stored as variables with a 'PARM extension. The *:ctype* of the constant must be the type of this special variable.

<DELAY> ==>
The *:type* of the expression must be INTEGER i.e. the Lisp list ('INTEGER). (Type specifications are always given as a list.)

<EXTERN_FUNCTION> ==>
The type-specifier must be an existing type i.e. the result of sending *:ptype* may not be equal to NIL. A function called **type-p** is defined to check whether this is true and will colour this specifier if it makes no sense.

<FORTRAN_FUNCTION> ==>
See <EXTERN_FUNCTION>.

<FORWARD_FUNCTION> ==>
The information under the key *"ident"* is looked up. This must be a 'normal' function.

<FOR_DOWNTO_DO_STAT> ==>
Correct assignments are given by table J.2.

<FOR_TO_DO_STAT> ==>
See <FOR_DOWNTO_DO_STAT>.

<FUNCTION> ==>

If a forward functions exists with the same name, then the parameters and the type-specifier must be coloured if they exist. The type-specifier must have a *:ptype* result or else it must be coloured. Double forward, extern and fortran functions are checked by using a symset object in which the routine-names are stored.

<FUNCTION_APPL_WITH_PARS> ==>

A function and its parameters can be found in their forward or normal declaration. The length of the parameter list and the list in the functions application must be equal. If so, then correct types must be used. This is checked by means of mapping the function **type=** on the two parameter-lists. The two compared types must be equal or an INTEGER value may be assigned to a REAL parameter. If the type-specifier is of kind 'VAR, which means that an expression may be assigned to this parameter, then the assignment is correct. If not, then the parameter must be a variable. A method called *:vartype* is defined to check this. If an expression is equal to a variable then it returns the Lisp symbol 'VARIABLE. The *:vartype*-methods are defined in [7].

<GOTO_STAT> ==>

The integer jumped to must be used in some kind of labelled statement. It will therefor be present in a defined symlayer. The key, used to look up the 'ACTUAL-label, is (label-value 'ACTUAL). A warning is given to the user to dissuade *GOTO-statement* usage.

<HHDL_PROGRAMS> ==>

Double program names are checked by means of the definition of a symlayer object. The information stored is ("ident" 'MODULE) or ("ident" 'PACKAGE). This allows package- and module names to be looked up by e.g. a use-statement evaluator.

<IDENT_LIST_IDENT> ==>

Checking if the <IDENT> in this structure is a type-specifier is done by means of the function **type-p** which sends the message *:ptype* to this object. Only type-specifiers respond to this message.

<IF_THEN_ELSE_STAT> ==>

The type of the expression in the if-then-else statement is checked by means of the function **boolean-p** which colours its argument if it is not of type BOOLEAN.

<LABELLED_STAT> ==>

The label-value used in this statement must be defined in the declarations of the program. The function **check-decl-label-p** is used to look up the information under the key *label-value*. A declared label is stored as (label-value 'DECLARED).

<MODEL_LABELLED_STAT> ==>

See <LABELLED_STAT>.

<MODULE> ==>

All comptype names are stored in a symset to realize double comptype and translator name checking. Double subprocess names, double forward, extern and fortran functions are checked in the same manner.

<MULT_FACTOR> ==>

The allowed types of the subobjects of this object are given by appendix G.

<NETASSIGN_STAT> ==>

Net identifiers in this object must be pins and may not be of kind 'INWARD. This is checked by the **not-in-pin-p**-function. The assignment of types by the <NETASSIGN_STAT> is checked by mapping the function **compare-types** on the elements of the net-id-list and the indexes-list. Those pairs must be of the same *net*type. The length of both lists must be equal. Pins may be used only once in the id-list.Double occurrences are checked by means of the definition of a symset storing the used pinnames.

<NETTYPE_DEF> ==>

The information belonging to the nettype-specifier is looked up in the tree. The kind of information retrieved must be a 'NETTYPE.

<NOT_FACTOR> ==>

The type of the subobject FACTOR is checked by means of the function **boolean-p**.

<PACKAGE> ==>

Forward, extern and fortran function names are stored in a symset to realize double declaration checking.

<PROCEDURE> ==>

Parameters in a procedure declaration may only be present if the procedure is not forward declared. This is checked by means of the function **check-forward-exist**. This function will colour parameters if a forward declaration exists.

<PROC_STAT> ==>

See <FUNCTION_APPL_WITH_PARS>.

<REL_EXP> ==>

The allowed subobject-types are given in appendix G.

<REPEAT_UNTIL_STAT> ==>

The expression type is checked by means of the function **boolean-p**.

<SIGNAL_STAT> ==>

The identifier object given must be of type semaphore. A function **semaphore-p** which checks the type and colours objects if desired.

<SIGNED_IDENT> ==>
The <IDENT> object must be a constant, i.e. it must respond to the message *:ctype* and it must of type INTEGER or REAL.

<SIGNED_TERM> ==>
The *:type* response of the subobject <term> must be INTEGER or REAL if the *:value* of the <sign> is not NIL, i.e. if the optional sign is used.

<SUBP_LABELLED_STAT> ==>
See <LABELLED_STAT>.

<SYNC> ==>
The **clock-type-p**-function is used to check if the used identifier is a *clock*. The expression is checked by means of the **integer-p** function.

<TIME_UNIT_STAT> ==>
The phil integer representation is converted to a Lisp value by using the lisp function **read-from-string**. This value must be within the range 0-15.

<TRANSMIT_TO> ==>
The *:type* of the expression is compared to the *:pintype* of the variable. A *:pintype* defined for the variable checks if the variable is a PIN. The function **nettype-info** looks up a type-specifier and checks if it is a nettype specification.

<UPON_DO> ==>
The function **boolean-p** is used to check and colour the expression.

<VAR_IDENT_LIST_IDENT> ==>
The function **type-p** is used to check if the identifier is a type-specifier.

<WAITFOR_STAT> ==>
The function **boolean-p** is used to check and colour the expression.

<WAIT_STAT> ==>
The function **semaphore-p** is defined to check if an object is of type SEMAPHORE. Its argument will be coloured if this is not the case.

<WHILE_DO_STAT> ==>

The function **boolean-p** is used to check and colour the expression.

<WITH_STAT> ==>

The variables referred to in the statement-list of a *with-statement* are the fields of the record-variables listed in the *rec-var-list* of this statement. In order to acces such fields a list of field-variables is determined. The method *:make-rec-list* creates a special list containing the fields and their types of the RECORD variables given by the *rec-var-list*. A *check-rec-list*-field is defined for the *with-statement* which is equal to this variable-list. A *field*-variable can be retrieved by subobjects of *<with_stat>* by sending the message *:lookup-rec-field* up the tree. The field is created by appending all the *field-type*-specifications of the record-variables given. These specifications are stored in the local lisp variable *rec-list*. The fields of the first record-variable of the *rec-var-list* are stored first. The next record-variable of the *rec-var-list* may be a sub-field specification of the first record-variable or an other record-variable.

Allowing just one DEFAULT clock ==>

All *:values* of the *clock_defs* (the optional strings "DEFAULT") are stored in a symset-object. Double occurrences are are coloured automatically.

Checking Standard Functions and Procedures

<CURRTIME> <RECALL> <LASTTIME> ==>

A function **check-pin-type** is defined to check if the argument is a PIN. The *:type* of the argument must be of type 'PIN.

<EOLN> <EOF> <RESET> <REWRITE> <PAGE> ==>

A function **txt-p** is defined to check if the *:vtype* of the argument is of type 'TEXT.

<ABS> <SQR> <SQRT> <SIN> <COS>
<ARCTAN> <EXP> <LN> <ODD> <TRUNC> <ROUND> ==>

The function **intreal-p** is used to check if the function argument is an integer or a real. The *:type* of the argument must be a member of the list INTREAL i.e. the list '(INTEGER REAL).

<ORD> <CHR> <SUCC> <PRED> ==>

The function **ordinal-p** checks if the *:type* of the argument is an INTEGER, CHARacter, BOOLEAN , an ENUMERATED or a SUBRANGE type.

<NEW> The first argument of the NEW routine must be a pointer type variable. This is checked by the function **pointer-p**.

The remaining arguments must be variant field specifiers of a RECORD type variable. A function *check-variant-label* is defined to check the variant fields recursively.

<READ> <READLN> ==>

The first argument may be a 'TEXT variable. The type of all argument must be INTEGER, REAL, CHAR or subranges of INTEGER, REAL, CHAR. This is checked by the function **read-var-type-p**.

<WRITE> <WRITELN> ==>

The types of the arguments of these routines must be a member of list '(INTEGER REAL CHAR STRING BOOLEAN), an ENUMERATED or a SUBRANGE type argument. This is checked by the function **write-exp-p**. The first argument of <WRITE> or <WRITELN> may however be a 'TEXT variable.

Appendix K

default-plane

Bibliography

[1] Phil Definers manual version 2.0

[2] Phil House Style

[3] Documentation of the PHIL package. (userman.txt)

[4] Specifying user interfaces Version 2.0 .

[5] Helix Command Reference Manual, Volume 2

[6] Helix Environment User Manual

[7] Helix Environment Specification in Philgram